DRIVER: UNA PLATAFORMA DE ECONOMÍA COLABORATIVA

DRIVER: A SHARING ECONOMY PLATFORM

UNIVERSIDAD
COMPLUTENSE
MADRID

TRABAJO DE FIN DE GRADO

Grado en Ingeniería de Software

Año 2017-2018

Autor

Rubén Casado Domínguez

Directores

Adrián Riesco Rodríguez

Enrique Martín Martín

RESUMEN

La economía colaborativa se define como una interacción entre sujetos a través de un medio digitalizado que permite satisfacer las necesidades de ambas partes. Es un paradigma que ha crecido exponencialmente en estos últimos años, siendo aplicada a todo tipo de servicios, i.e. alojamiento, conocimientos, objetos de segunda mano o transporte.

El objetivo de este trabajo es el desarrollo de una aplicación web, lo que implica una prueba de la implementación de este tipo de aplicaciones. La aplicación está fuertemente ligada al concepto de economía colaborativa aplicada al transporte, y se ha elaborado mediante herramientas como Maven, AngularJS, Spring, Bootstrap y Servicios Web REST.

La plataforma permite compartir vehículo con otros usuarios interesados en hacer la misma ruta. La moneda de intercambio para estos servicios se basa en el tiempo empleado en hacer los viajes.

Palabras Clave

Maven, AngularJS, Servicios Web, Economía Colaborativa, Carpooling, *Back-end, SPA, API Google Maps, Front-end, Spring.*

ABSTRACT

The sharing economy concept can be described as an interaction between users that allows them to satisfy their needs on both parts, using a digital platform. It has been a growing paradigm in the most recent years and has spread to all kind of services, i.e. accommodation, knowledge, second-hand objects or conveyance.

The aim of this project is to develop a web application, which entails the implementation of this kind of service. This app is strongly linked to the concept of sharing economy, and has been ellaborated with tools as Maven, AngularJS, Bootstrap and REST Web Services.

The web application allows users to share their vehicle with other people interested in the same way. The coin used to pay for this services is the time spent during the trips.

Key Words

Maven, AngularJS, Web Services, Sharing Economy, Carpooling, Back-end, Front-end, SPA, API Google Maps, Spring.

ÍNDICE

ÍNDICE DE FIGURAS

1. INTRODUCCIÓN

El ámbito tecnológico se encuentra en una situación de cambios constantes, con el objetivo de mejorar las condiciones de vida de la humanidad. Dentro de las organizaciones profesionales y las empresas, las áreas dedicadas a las tecnologías deben trabajar en las tareas diarias de elaboración de los productos y desarrollo orientado a sus clientes, optimizando la calidad y el soporte que ofrecen. Sin embargo, para crecer como empresa, se deben encontrar soluciones que permitan trabajar con herramientas y espacios adaptados a las singularidades de los empleados. Es por ello que las empresas invierten gran cantidad de dinero en mejorar sus aplicaciones web, el entorno de trabajo común para muchos de los empleados.

Cada vez son más las aplicaciones que permiten servicios basados en economía colaborativa, lo que facilita la vida de muchas personas, además de incentivar la comunicación y el espíritu de comunidad entre ellas. Como se ha mencionado anteriormente, el auge de la economía colaborativa, ha permitido crear infinidad de aplicaciones de este tipo, hay algunas que proponen la compartición de vivienda, otras intercambiar objetos, ofrecer alojamiento y comidas a cambio de ayudas a los dueños y, por supuesto, compartir el propio vehículo para aprovechar al máximo la ruta, como se propone en este proyecto.

Esta aplicación nace con el objetivo de crear una plataforma que permita a sus usuarios poder transportarse con un medio de pago justo, como el tiempo. Se intercambia un servicio que implica el mismo esfuerzo para las dos partes implicadas. Además, este concepto implica una reducción en la contaminación medioambiental, ya que ayuda a utilizar el mínimo de vehículos posible transportando las mismas personas.

Para ello, lo primero que se ha hecho es una planificación temporal, contando con el tiempo disponible, que realmente está, desde un principio, muy ajustado para realizar todas las funcionalidades pensadas. En primer lugar, se parte con una planificación que va desde marzo hasta mediados de septiembre, lo que da un margen de 6 meses y medio, lo que implica poco tiempo debido a que el alumno también tiene otras asignaturas por cursar y, a parte, un trabajo laboral diario.

1.1 Planificación temporal

En el mes de **marzo**, el trabajo a realizar es la elección del proyecto y especificación de las funcionalidades que debe contener, como son el registro y la autenticación a la aplicación, la publicación de nuevos viajes, la unión a viajes de otros usuarios y un buzón de mensajes.

Durante el mes de **abril**, una vez especificadas las funcionalidades que debe contener el proyecto, se pasa al diseño de la base de datos y se comienza con el diseño de las interfaces de usuario que debe contener la aplicación, haciendo bocetos de cada una de las ventanas.

En **mayo** se dedica la mayor parte del tiempo al aprendizaje de las nuevas herramientas haciendo pruebas antes de empezar con el desarrollo, también se especifica la estructuración y organización del código fuente.

En los meses de **junio, julio y agosto**, se dedica todo el tiempo disponible a desarrollar el código de la aplicación, siendo a mediados de agosto cuando ya hay suficiente código desarrollado como para dar comienzo a la documentación de la memoria, lo que dará lugar a una combinación de trabajo entre programación y documentación. Aunque ya la mayor parte de la programación en esta última mitad de mes se basa en la realización de pruebas y resolución de errores.

Por último, en el mes de **septiembre**, se entrega el trabajo realizado a los tutores del alumno, siendo estos los que ordenan las últimas pinceladas a la memoria del proyecto. La cual debe de ser totalmente terminada a mediados de este mes.

1.2 Estructura de la memoria

El resto de la memoria se divide como sigue:

- En la sección 2 se detallan los preliminares, es decir, se citan las tecnologías usadas más importantes y las técnicas utilizadas.
- En la sección 3 se describen los componentes de la aplicación y la interacción entre ellos detalladamente.
- En la sección 4 se presentan las conclusiones finales, contando con la valoración crítica del alumno y el trabajo futuro que podría tener este proyecto.

1.3 Link del repositorio del código

https://github.com/RubenK01/driverBoot

1. INTRODUCTION

The tech world is in a constant situation of changes, aiming the improvement of human beings' life. Within the companies and profesional organisations, tech departments work everyday in product ellaboration and development, always client-oriented.

Nevertheless, useful tools and spaces adapted to the employees must be found in order to grow as a company. That is why there's a high investment in improving companies' web applications, the common environment for all employees.

Everyday new sharing economy applications appear, which are always a way to ease people's life and encourage communication and comunity spirit.

This app emerges with the purpose of creating a platform to get conveyance paid with a fair method: time. The service exchanged requires the same effort for both parts.

The idea promotes using the least possible cars, impliyng also a reduction in environmental pollution.

The first step to achieve this goal was to set up a timing planification. The available time to make up all the planned features was tight from the beginning.

The timing went from March to September being 6 months and a half for the app development, which is a narrow window given the fact that the student has a part-time job and had other assignments in university.

1.1 Timing

On **March**, the project had to be chosen and defined, as well as the features: sign up, log in, new trip posting, joining trips and a message box

On **April**, design of database and interfaces were the main planned activities. Sketches of every window were made.

May was reserved to learn about the tools, testing them before starting the ellaboration of the code. Structure and organization of the source code was also defined.

On **June, July and August** all the available time was dedicated to development. In mid august the app was almost finished, from then on the student started writing this paper as well as testing and fixing errors.

Last but not least, on **september** the first draft was handed to the tutors, so they could determine the final changes.

1.2 Structure

The following chapters of this paper are:

- Section 2: Preliminaries. Most important used tools and techniques are described here.
- Section 3: Description. The components and interactions of the app are reported, and depicted by screenshots.
- Section 4: Conclusions and future work. Contains a critical analysis of the student and future perspectives to make improvents of the application

1.3 Link to the repository

https://github.com/RubenK01/driverBoot

2. PRELIMINARES

En este apartado se presentan algunos conceptos necesarios para comprender el contenido de la memoria. Así, en la sección 2.1 se comenta cómo surgió la idea y los motivos de la creación de este proyecto. En la sección 2.2 se exponen las tecnologías y técnicas utilizadas en el proyecto.

2.1 Presentación

En primer lugar, se describen los dos conceptos más importantes de los que nace la idea de este proyecto: **"*carpooling*"** y ***"banco de tiempo".***

Por un lado, ***carpooling*** es la práctica que consiste en compartir un automóvil particular con otras personas. Esta práctica fomenta una disminución de contaminación, así como un ahorro económico para sus usuarios, ya que supone compartir el gasto del combustible. Además, permite a los usuarios llegar en automóvil a zonas de difícil aparcamiento, así como zonas a las que no se podría llegar en transporte público. Otra de las grandes ventajas de esta práctica son las relaciones sociales entre los usuarios.

Forma parte de la economía colaborativa, ya que se busca colaboración entre particulares para conseguir un objetivo común: reducir gastos. En 2011, DongWoo y Liang (1) propusieron un modelo de *carpooling* en el cual se generan recomendaciones para los usuarios que tienen rutas habituales en común, mediante localización GPS.

Por otro lado está el **banco de tiempo**, un sistema que permite a sus usuarios prestar un servicio a cambio de tiempo. Gracias a este tiempo ganado por el usuario que presta el servicio, más adelante podrá recibir un servicio de otro usuario equivalente al tiempo que ha prestado. La idea pretende reducir la necesidad de dinero convencional, facilitando a los individuos de la comunidad el acceso a ciertos recursos, así como reducir su dependencia del dinero y mercado tradicional.

Este sistema permite, por una parte, un comercio justo, ya que se va a invertir el mismo esfuerzo que el que se va a recibir, en cambio con dinero, esta premisa pocas veces se

cumple. Por otra parte, se permite a usuarios con pocos recursos económicos recibir servicios a los que no tendrían acceso si se midieran con dinero, lo que pone en igualdad los distintos estratos económicos. (2)

Un ejemplo es el tándem lingüístico, en el que se produce una relación entre dos individuos de habla nativa distinta, y que intercambian conocimientos. Es decir, cada uno enseña al otro su lenguaje.

Una vez explicados estos dos conceptos base, se presenta el proyecto **"*Driver*"**, el cual nace como una mezcla de estas dos ideas. Esta aplicación tiene como objetivo principal juntar usuarios que vayan a realizar la misma ruta para aprovechar al máximo el uso del vehículo y combustible.

De esta forma, por una parte, los pasajeros recibirán un servicio "gratuito" en cuanto a dinero, además de poder socializar con el resto de los usuarios y llegar a acuerdos con ellos en caso de ser un trayecto habitual, como por ejemplo, personas que van todos los días a trabajar a la misma zona. A su vez, se evita la contaminación que causaría que cada una de estas personas que hacen el mismo recorrido cogiesen su vehículo.

Por otra parte, el conductor recibirá los minutos que se tarda en hacer el recorrido (calculados automáticamente por la API de Google Maps), multiplicado por los pasajeros que se hayan unido al viaje. Así, después de este recorrido, este usuario podría unirse a otro viaje que le pueda interesar y consumir sus minutos acumulados.

Resumiendo, este proyecto pretende aprovechar los beneficios que tiene el *carpooling* con los beneficios del *banco de tiempo.* Y todo esto, gracias a *Driver* se podrá gestionar con una aplicación web utilizando las últimas tecnologías, lo que permite estar al alcance de cualquier persona y desde prácticamente cualquier plataforma, ya que cuenta con una agradable y sencilla interfaz *responsive,* lo que significa que se adapta a cualquier tamaño de pantalla.

2.2 Tecnologías y técnicas utilizadas

En esta sección se dan a conocer las tecnologías y herramientas usadas a lo largo del proyecto, junto con un breve resumen de cada una. El proyecto se desarrolló sobre un ordenador personal, el cual cuenta con unas especificaciones básicas (procesador Intel Core i5 y 8Gb de RAM), sin ningún otro soporte hardware.

En primer lugar, se tomó la decisión de realizar el proyecto sobre una aplicación web, ya que cuenta con grandes ventajas sobre las aplicaciones de escritorio:

- No requieren instalar software especial: Para acceder a ellas es suficiente con disponer de un navegador web, que, al menos uno, suele venir instalado con el sistema operativo.
- Bajo coste en la actualización: Los navegadores web visualizan las páginas web que son accedidas a través del servidor web dinámicamente. Por lo que es el servidor el que ejecuta la mayor parte de código y suministra las vistas de las páginas a los navegadores. En consecuencia, las actualizaciones se hacen directamente en el servidor, y automáticamente se verá en todos los navegadores.
- Acceso a la última y mejor versión: Con lo explicado en el punto anterior, se evita que algún equipo funcione con una versión desactualizada.
- Movilidad: El software está ubicado en un servidor web en Internet y, por lo tanto, cualquier usuario con un dispositivo adecuado y una conexión a internet, podría acceder a una aplicación web.
- Reducción de costes en los puestos del cliente: Debido a que las páginas se ofrecen desde un servidor web, no es necesario un hardware potente en la máquina del cliente, lo que se traduce en reducción de costes y una mayor longevidad.

Cabe destacar que la mayoría de las herramientas, antes de este proyecto, eran conocidas por el alumno a un nivel básico, y algunas de ellas totalmente nuevas, como por ejemplo las APIs de Google e integración de estas APIs en AnguarJS, algo con bastante poco soporte en Internet.

2.2.1 LENGUAJES Y HERRAMIENTAS SOFTWARE

A continuación, se presentan los lenguajes utilizados en el proyecto y las herramientas o bibliotecas software, así como su aportación en el proyecto.

Como cualquier otra aplicación web, se divide en dos partes: el *Back-end o Servidor* y el *Front-end o Cliente.*

2.2.1.1 Back-end / Servidor

Apache Maven Project [ver. 4.0.0] (3)

Es una herramienta de código abierto creada en 2001. Su principal característica es la simplicidad para compilar y generar ejecutables como JAR, WAR, etc. Con un simple comando, independientemente de cada proyecto con sus diferentes dependencias, es posible generar la *build*.

Además de esto, Maven permite gestionar fácilmente todas las dependencias de módulos y distintas versiones de bibliotecas del proyecto, así como su actualización automática gracias a su repositorio remoto *Maven Central*. Tan solo habría que incluir estas dependencias en el fichero XML de configuración de Maven llamado POM[1].

Maven aporta una estructura de proyecto muy intuitiva y sencilla para que cualquier nuevo desarrollador pueda entender rápidamente el código, lo que permitirá construir *software* bien organizado y de calidad. Más adelante se expone la estructura utilizada en este proyecto.

[1] **POM.xml**: Project Object Model es un fichero XML que es la "unidad" principal de un proyecto Maven. Contiene toda la información del proyecto, como fuentes, dependencias, tests, plugins, versiones, etc.

También es capaz de gestionar un proyecto desde la etapa de validación hasta el despliegue, es decir, permite pasar test, empaquetado, pruebas de integración, verificación y despliegue de la aplicación en su entorno.
Para proyectos de mayor envergadura con dependencias en otros proyectos, Maven también permite gestionar dichas dependencias.

Java Spring [ver. 1.2.3] (4)
Es un potente *framework* de Java, el cual es muy amplio y dispone de innumerables funcionalidades.
Java es uno de los lenguajes de programación orientado a objetos más populares y completos que existe a día de hoy, por lo que cuenta con una gran comunidad de usuarios. Además, Spring se complementa a la perfección con Maven, lo que hace ideal su elección para este proyecto.
Entre las funcionalidades que ofrece Spring, las más importantes de cara a este proyecto son las siguientes:

- *Servicios REST* es una de las mejores funcionalidades de Spring, ya que a través de algunas etiquetas como *@RestController* para indicar el método/objeto que implementará el controlador REST de la aplicación, es sencillo manejar las peticiones REST. También permite recoger el *body* de las peticiones POST y manejar los parámetros de las peticiones GET. Esta funcionalidad maneja cualquier tipo de petición, como GET, POST, PUT y DELETE.
- *Spring Security* (5) gracias a esta funcionalidad, se facilita la configuración del inicio y cierre de sesión. Spring se encarga en cada petición de comprobar si un usuario esta registrado y logueado[2]. Además, en este proyecto también se ofrece una capa de protección extra al encriptar todas las contraseñas de los usuarios con la biblioteca ***BCryptPasswordEncoder***, de la cual se hablará más adelante. *Spring Security* necesitará de algunos archivos de configuración en los cuales se describirán todas las opciones personalizadas, como por ejemplo, cuál será la página de inicio de sesión, qué se debe hacer si el usuario intenta acceder a la

[2] **Loguear:** Anglicismo proveniente de la palabra Login que se refiere al hecho de iniciar sesión en una aplicación.

aplicación sin estar logueado, permisos a ficheros dependiendo del rol del usuario, etc.

Esta capa de seguridad incluida por Spring utiliza el estándar (RFC 7519) ***JWT*** [3]***.***

- *Spring Data JPA* es una importante funcionalidad de Spring que permite usar ***JPA (Java Persistence API)*** en nuestro proyecto Java Spring. Es decir, Spring otorga todas las funcionalidades necesarias para poner JPA en práctica. Por ejemplo, permite declarar las entidades mediante la etiqueta *@Entity(name="nombre")* las cuales conectarán directamente a la base de datos y usarán ***Hibernate***[4] para el mapeo de sus atributos entre la base de datos relacional y el modelo de objetos de la aplicación. Otra funcionalidad importante es la que nos permite ejecutar acciones en la base de datos mediante la etiqueta *@Repository*, donde existirán funciones por defecto de JPA como *save*, *findAll*, etc. Y también se podrán crear nuevas funciones gracias a la etiqueta *@Query(consultaSQL).*

- *Spring Bean* es un objeto configurado e instanciado en el contenedor de Spring, lo que permite el acceso a este durante la ejecución desde cualquier parte de la aplicación. Spring está basado en este concepto de *beans* almacenados para implementar la mayoría de sus características, como la inyección de sus dependencias.

  Esta característica ha resultado verdaderamente útil para el desarrollo de la aplicación, ya que la etiqueta *@Service* permite instanciar un nuevo *bean,* en el cual se pueden configurar los formatos con los que se quieren recoger los datos del repositorio. Se accede automáticamente desde cualquier controlador o clase de la aplicación simplemente añadiendo la etiqueta *@Autowired* a nuestro *bean*.

[3] **JWT:** JSON Web Token es un estándar abierto basado en JSON utilizado para la creación de tokens de acceso.

[4] **Hibernate:** es una herramienta de mapeo objeto-relacional (ORM) para la plataforma Java que facilita el mapeo de atributos entre una base de datos relacional y el modelo de objetos de una aplicación, mediante archivos declarativos (XML) o anotaciones en los *beans* de las entidades que permiten establecer estas relaciones.

BCryptPasswordEncoder

Es un *bean* de Spring, el cual permite encriptar las contraseñas usando el algoritmo *BCrypt*, una función *hash* basada en el algoritmo de cifrado de bloques simétricos llamado *Blowfish*[5]. Sin embargo, *BCrypt* añade un nuevo factor de seguridad, este algoritmo genera un valor *salt*[6] totalmente aleatorio, lo que hará que en cada llamada se genere un valor de *hash* distinto. Esto hace mucho más costoso un ataque de fuerza bruta para descifrar una contraseña. Además, *BCrypt* genera una cadena de longitud 60, lo que le dará aún más seguridad y lo hace ideal para la encriptación de las contraseñas de esta aplicación web.

JPA (Java Persistence API) (6)

Se trata de un API de Java que permite implementar el almacén y los objetos del dominio, lo que hará mucho más sencilla y visual la manipulación de los datos.

Cada entidad representa un objeto del negocio persistente, lo que en la base de datos se suele traducir por una tabla. Esta entidad puede tener entre sus atributos alguna conexión con otra entidad, por ejemplo, en este proyecto hay una entidad 'Usuario' que se conecta a otra entidad 'Coche'. Así, bastaría con añadir una etiqueta en el que se indica a JPA con qué atributo de la otra entidad se tiene que mapear y su cardinalidad. En las figuras 2.1 y 2.2 se muestran las etiquetas de la entidad Usuario y Coche respectivamente, a modo de ejemplo.

```
@OneToMany(cascade= CascadeType.ALL)
@JoinColumn(name="conductor")
private Collection<Coche> coches;
```

Figura 2.1 Atributo "coches" de la entidad "Usuario"

```
@ManyToOne
private Usuario conductor;
```

Figura 2.2 Atributo "conductor" de la entidad "Coche", referenciado en la entidad "Usuario"

[5] **Blowfish:** es un codificador de bloques simétricos, diseñado por Bruce Schneier en 1993 e incluido en un gran número de conjuntos de codificadores y productos de cifrado.
[6] **Salt:** comprende bits aleatorios que se usan como una de las entradas en una función derivadora de claves.

JPA hace uso de una clase llamada *EntityManager* la cual permite conectarse al repositorio de datos para hacer consultas que ya tiene por defecto la API, como *save, find, findAll, etc.* También se pueden crear nuevas consultas personalizadas.

Además JPA cuenta con un fichero de configuración, denominada *application.properties* en este proyecto. Este fichero debe contener todas las variables necesarias para la conexión a la base de datos.

En este fichero, también se puede especificar a JPA con el fin de crear las tablas de la base de datos automáticamente. Esto se suele hacer la primera vez que se ejecuta la aplicación; después simplemente se hará un *update* a la base de datos por si hubiera algún cambio en las tablas que se deba actualizar sin hacer un borrado de los datos anteriores.

Bases de Datos Relacionales con MySQL (7; 8)

Una base de datos relacional es aquella que representa los datos y las relaciones mediante tablas, cada una con un nombre único, donde cada una de las filas de la tabla representa una relación entre un conjunto de valores.

Este concepto fue definido por Edgar Frank Codd a finales de los años 60, quien más tarde publicó el documento más importante sobre esta materia llamado '*A Relational Model of data for Large Shared Data Banks'* (9). El modelo de datos relacional consta de 3 características fundamentales: Estructura de datos, Integridad de los datos y Manipulación de los datos.

También se ha tratado de normalizar la base de datos hasta la segunda forma normal (2FN), sin llegar totalmente hasta esta, ya que podría suponer una falta de claridad sobre la base de datos y, además, conllevaría un gasto de tiempo del que no se dispone. La **normalización** pretende solucionar varios problemas como: ambigüedades, redundancia, pérdida de restricciones de integridad y anomalías en operaciones de modificación de datos.

Para gestionar la base de datos relacional en este proyecto se ha utilizado el *SGDB* (Sistema de Gestion de Bases de Datos) **MySQL**, considerada la base de datos de código abierto más popular del mundo.

Una gran ayuda para la gestión y visualización de esta base de datos ha sido el *server* ***XAMPP***, un servidor que consiste principalmente en la base de datos MySQL, servidor Apache y los intérpretes para lenguajes de *script*: PHP y Perl. Para este proyecto se han utilizado los módulos de MySQL para la base de datos y Apache para la visualización de esta base de datos a través de phpMyAdmin, cuya interfaz se presenta sencilla de utilizar y contiene todo tipo de utilidades necesarias para la gestión de la base de datos.

Técnicas y Patrones de Diseño

Cabe destacar que este proyecto incluye la implementación de varios patrones de diseño aprendidos a lo largo del grado universitario cursado por el alumno (sobre todo, en las siguientes asignaturas: Aplicaciones Web, Ingeniería del Software y Modelado de Software).

A continuación, se citan junto con una breve descripción algunos de los patrones más importantes usados en la aplicación (10; 11):

- *DTO (Data Transfer Object).* Este patrón permite independizar el intercambio de datos entre la capa del cliente y el modelo de datos. Así, al cliente sólo le llegará la información requerida con el formato deseado. Permite mover múltiples datos entre capas utilizando un único objeto.
- *DAO (Data Access Object)*. Permite encapsular en un objeto todas las llamadas a la base de datos, junto con sus consultas correspondientes. Con esto se pretende, por un lado, desacoplar la capa de persistencia de la capa de negocio y, por otro lado, liberar al resto de capas de información innecesaria como pueden ser sentencias SQL o información para la conexión con la base de datos. El patrón DAO en este proyecto se implementa gracias a JPA.
- *BO (Business Object)*. Los objetos de negocio permiten albergar en la capa de negocio una representación de las entidades contenidas en la base de datos, con los atributos y relaciones pertinentes. Gracias a JPA se pueden añadir ciertas etiquetas a los atributos, como se ha mencionado anteriormente, para crear y configurar automáticamente las relaciones entre las entidades. Tratar las entidades como objetos facilita en gran medida la gestión de los datos que

además, al tratarse de objetos Java, también podrían contener cierta lógica. Asimismo, se evita que componentes de la capa de negocio accedan directamente a la capa de persistencia.

- *Facade* permite implementar una interfaz simple entre la capa del cliente y la capa de negocio. Así, se abstrae al cliente de los cambios futuros que puedan existir, lo que ayuda a desacoplar capas. En esta interfaz, no se debería desarrollar apenas lógica, sino hacer llamadas a otras clases las cuales sí que desarrollarían la lógica deseada.

- *Application Service*. Este patrón permite implementar la lógica sobre los objetos de negocio que el patrón *facade* explicado anteriormente no alcanza. Así, gracias a este patrón se pueden desarrollar las reglas de negocio y formatear los DTO para mover la información a los Objetos de Negocio, los cuales más adelante conectarán con el repositorio de datos para realizar las operaciones oportunas.

- *MVC (Model, View, Controller).* Es un patrón arquitectónico de *software* que propone separar el código de los programas por sus diferentes responsabilidades. Es decir, separa los datos y la lógica de los eventos y las vistas. Se divide en tres componentes distintos: Modelo, Vista y Controlador.

  Tiene diversas ventajas como la **organización** del código, ubicando los distintos tipos de lógica según su finalidad, lo que facilita el mantenimiento y la escalabilidad del código, **sencillez** para crear distintas representaciones de los mismos datos y **reutilización** de los componentes.

2.2.1.2 Front-end / Cliente

Bootstrap [ver. 3.3.5] (12)

Se trata de una biblioteca multiplataforma de código abierto que tiene como principal objetivo el diseño web. Ofrece diversas plantillas para formularios, botones, tablas, menús, etc.

Además ofrece, jugando con hojas de estilo o *CSS y Javascript,* una estructura en la que se puede dividir el ancho del contenedor web en 12 partes iguales, lo que permite

una gran facilidad para organizar el contenido de una página web. Además, puede configurarse la adaptación del contenido a distintas resoluciones de pantalla, como por ejemplo un móvil o una *tablet*. A parte de poder organizar el contenido según el ancho de un contenedor web, también se puede organizar por filas o *rows*.
Son infinitas las utilidades para diseño *front-end* que puede aportar esta biblioteca. Gracias a *Bootstrap* esta aplicación puede abrirse y usarse desde un móvil o *tablet* sin perder ningún detalle.

JavaScript (13)

Es un lenguaje de programación interpretado, dialecto del estándar *ECMAScript*. Se utiliza principalmente en el lado del cliente, ejecutado directamente por el navegador web, lo que permite mejorías en la interfaz del usuario y páginas web dinámicas.
Para interactuar con una página web se provee a JavaScript de una implementación del *Document Object Model (DOM)*, lo que proporciona el uso de los elementos contenidos en el documento HTML.
Gracias a este lenguaje se puede ejecutar cierta lógica en el cliente (navegador web), aunque no debe ejecutarse toda la lógica del negocio, ya que es fácil sobrecargar el navegador y provocar bloqueos, por lo que el grueso de la lógica debería de estar alojado en el servidor.
En cuanto a la estructura de la programación, se puede decir que es compatible casi en totalidad con la estructura de C, como por ejemplo es igual para sentencias *if, while, for, switch, etc,* pero, por ejemplo, en JavaScript no es necesario la declaración de un tipo específico de variable ni escribir punto y coma ";" al final de cada instrucción, a diferencia de C .

AngularJS [ver. 1.4] (14)

Anteriormente en la parte *front-end* de las aplicaciones web se utilizaba únicamente jQuery y otras bibliotecas como ayuda para el código JavaScript, pudiendo hacer manipulaciones sencillas o añadir efectos, pero no existía un patrón a seguir, lo que provocaba que con el tiempo el código cada vez se hiciera más complicado de manejar. Por este motivo surgió AngularJS, un *Framework* de JavaScript de código abierto para el desarrollo de aplicaciones web en el lado cliente, que viene de la mano

de Google y utiliza el patrón MVC (patrón de diseño que se utiliza en el proyecto) como ayuda para crear interfaces de usuario separando conceptos.

Esta biblioteca es capaz de leer ciertas etiquetas personalizadas del HTML y, gracias a estas, la vista obedece a las directivas y se pueden manipular los elementos que contienen las etiquetas. Es decir, Angular es capaz de manipular el HTML en tiempo de ejecución sin actualizar la página web.

Se crea con el fin de dar mayor funcionalidad al lenguaje HTML con directivas y atributos, de manera que permita vistas dinámicas con mayor eficiencia, y con ello aplicaciones **SPA** (*Single-Page Applications*), las cuales se exponen más adelante.

Una directiva es una funcionalidad creada anteriormente con el objetivo de realizar alguna acción sobre el HTML que la contenga, por ejemplo AngularJS cuenta con una serie de directivas nativas, algunas de las más importantes son:

- *Ng-App* sirve para arrancar la aplicación Angular, indicando el elemento raíz.
- *Ng-Controller* indica cuál debe de ser el controlador que manejará el contenedor de esta etiqueta.
- *Ng-Model* permite conectar el valor de un *input* al controlador de Angular establecido.
- *Ng-Click* con esta directiva se puede establecer cuál debe ser la función a ejecutar cuando se pulse el elemento marcado por la etiqueta.
- *Ng-Repeat* para iterar sobre una lista declarada en el controlador.
- *Ng-If* permite implementar la sentencia condicional *if*.

Además de estas existen muchas más directivas nativas de Angular. También es posible que el desarrollador cree otras directivas que vea necesarias para su aplicación. En esta aplicación, por ejemplo, el alumno tuvo que crear una nueva directiva para poder guardar y visualizar imágenes.

También cabe destacar la importancia de la palabra reservada ***$scope***[7], la cual sirve para declarar variables o funciones que puedan llamarse o visualizarse desde el código HTML.

[7] **Scope:** se trata del ámbito de los datos donde estamos trabajando en las vistas.

Aplicaciones SPA (Single-Page Applications)

Son aplicaciones o sitios web que caben en una única página con el propósito de dar una experiencia más fluida a los usuarios como una aplicación de escritorio. En estas, todos los códigos se cargan o bien de una sola vez, o cargan los recursos necesarios dinámicamente como lo requiera la página, de manera que se van agregando, normalmente como respuesta de las acciones del usuario, de manera que la página no tiene que cargar otra vez en ningún punto del proceso. Un ejemplo de aplicación SPA, conocido por todos, es GMAIL.

Con la utilización de las aplicaciones web SPA se consigue una mejor experiencia para el usuario, al tener menor tiempo de espera o latencia entre vistas sin recargar el navegador. Para alcanzar este propósito, en SPA se utiliza un único punto de entrada, normalmente el *index.html*. Con este sistema de plantillas, los contenidos estáticos proporcionados por el servidor son mezclados en el lado cliente para generar la vista.

Estilo arquitectónico REST

Es estilo arquitectónico REST se puede definir como una arquitectura *software* para sistemas hipermedia[8] distribuidos[9]. En sus orígenes, esta arquitectura estaba basada en unos sólidos conceptos, como: la utilización de identificadores estándar de nombramiento, separación de aspectos y simplicidad.

En este contexto, coincidiendo con el crecimiento en sus primeros años de vida, junto con el interés de hacer de la Web una novedosa plataforma de *marketing,* provocó la necesidad de establecer los principios que subyacían en ella. De esto se encargaría Roy Fielding en su tesis doctoral (15) en el año 2000, teniendo como resultado un nuevo estilo arquitectónico para sistemas hipermedia distribuidos, denominándolo REST (*Representational State Transfer*), del que la *World Wide Web (WWW)* es su principal instancia. En esta tesis se documentan una serie de restricciones de diseño, ajustándose

[8] **Hipermedia:** término que designa al conjunto de métodos o procedimientos para escribir, diseñar o componer contenidos que integren diferentes soportes, obteniendo un alto grado de interactividad con los usuarios.

[9] **Distribuido:** término que define a los sistemas cuyos componentes *hardware y software*, conectados a la red, comunican y coordinan sus acciones mediante mensajes, estableciendo la comunicación mediante un protocolo prefijado por un esquema cliente-servidor.

a las necesidades de un sistema hipermedia distribuido de gran escala: **escalabilidad** en las interacciones entre componentes, interfaces genéricas, despliegue **independiente** de componentes y diseño de componentes intermedios para **reducir la latencia** de las interacciones, reforzar la seguridad y **encapsular sistemas heredados**.

Este estilo arquitectónico hace enfasis en la semántica de los conectores, consiguiendo efectos como: cacheado y reutilización de interacciones, sustitución dinámica de componentes y procesamiento de acciones por parte de intermediarios en la comunicación.

Así pues, REST define como debe comportarse una aplicación web: red de páginas web que componen una aplicación, donde los usuarios se mueven por la aplicación mediante la selección de enlaces.

Google APIs (16)

Para este proyecto se han utilizado algunas APIs de Google, por ejemplo para mostrar los mapas, rutas, buscar la localización del usuario y autocompletar algunos *inputs* con lugares existentes.

Esta API, al ser completamente nueva para el alumno, ha sido posible incluirla gracias a los tutoriales de la propia web de Google dedicada a desarrolladores y aplicando un poco de ingeniería para acoplar el código a AngularJS.

Se ha usado el módulo *autocomplete* para lo que ha hecho falta añadir la biblioteca *places*. También se ha usado el módulo *DirectionsService y DirectionsDisplay* para mostrar rutas, y el módulo *Map e InfoWindow,* para mostrar el mapa y los *markers* con la información personalizada manejada desde AngularJS.

2.2.2 ENTORNOS DE DESARROLLO

Eclipse [ver. Jee Oxygen]

Se ha utilizado el IDE (*Integrated Development Environment*) Eclipse Oxygen para lanzar el servidor Java EE, ya que es el más conocido por el alumno hasta la fecha para desarrollar en Java. Cuenta con grandes ventajas, como un buen depurador, la posibilidad de instalar multitud de módulos, editor de texto con resaltador de sintaxis, etc. Estas ventajas unidas a la experiencia del alumno con este IDE de este entorno son los principales motivos por los que ha sido elegido.

Sublime Text [ver. 3.1.1]

Este editor de código ha sido elegido para el desarrollo de la parte *front-end*, ya que cuenta con una buena indexación para las búsquedas, es muy ligero y gratuito. Al fin y al cabo, para programar la parte *front* no es necesario ningún depurador, ya que todo el código se ejecuta en el navegador. Además, cuenta con varios *plugins* muy útiles, como por ejemplo *sublimerge,* que permite la comparación de ficheros.

2.2.3 CONTROL DE VERSIONES

Git

Para este proyecto se ha decidido que el controlador de versiones más adecuado sería Git, porque se adapta bien a Eclipse y es de los controladores gratuitos más conocido. Aunque dispone de una versión de pago para conseguir que el código sea privado, no ha sido necesario adquirirla para este proyecto. Por otra parte, existe GitHub, una plataforma para compartir repositorios git de manera sencilla, ya que aporta una interfaz de usuario para quienes tengan dificultades con el uso de comandos.

A lo largo del proyecto se han ido haciendo *commits* casi todas las semanas con los avances realizados. Así, si en algún momento se perdiese código o no se pudiera detectar algún error, bastaría con descargar el código del anterior *commit.*

3. DESCRIPCIÓN DE LA APLICACIÓN

En este capítulo se explicará, en el apartado 3.1 la base de datos, en el 3.2 la estructura de los directorios de la aplicación y, en el 3.3, los principales componentes, así como la interacción entre estos. El apartado 3.3 se divide en dos secciones, en la 3.3.1 se describirán los componentes del *back-end* y en la 3.3.2 los componentes del *front-end*.

3.1 Base de datos

A continuación, se explicarán brevemente las tablas utilizadas en la base de datos de la aplicación. Para entenderse mejor, en la figura 3.1 se muestra el diagrama ER (Entidad-Relación), en el que se puede observar la cardinalidad de las relaciones y los atributos de cada tabla.

- Usuario: se almacenarán todos los datos personales de los usuarios, incluyendo la contraseña (atributo *password*) que estará encriptada con el algoritmo BCrypt como ya se ha explicado anteriormente en la sección 2.2.1.1.
- Role: se guardará el rol asignado a cada usuario, por defecto todos los usuarios tendrán el rol *USER*, pero podría especificarse, por ejemplo, un rol *ADMIN* con el que se podría acceder y modificar ciertos datos que con el rol *USER* no se permite.
- Mensaje: en esta tabla estarán localizados todos los mensajes de los usuarios. Gracias al atributo *leído* se podrá saber si el mensaje ha sido leído por el receptor.
- Viaje: se almacenan todos los viajes publicados por los usuarios. Contiene información sobre los minutos que dura la ruta, la fecha, el coche asignado, etc.
- Mapa: esta tabla está ligada únicamente a la entidad *Viaje*, ya que contiene toda la información de la localización de cada viaje en el mapa, como las coordenadas y la descripción de los puntos de origen y destino.
- Coche: en esta tabla se encuentran todos los coches de cada usuario, conteniendo información como la matrícula, modelo, color, foto y el id del propietario.

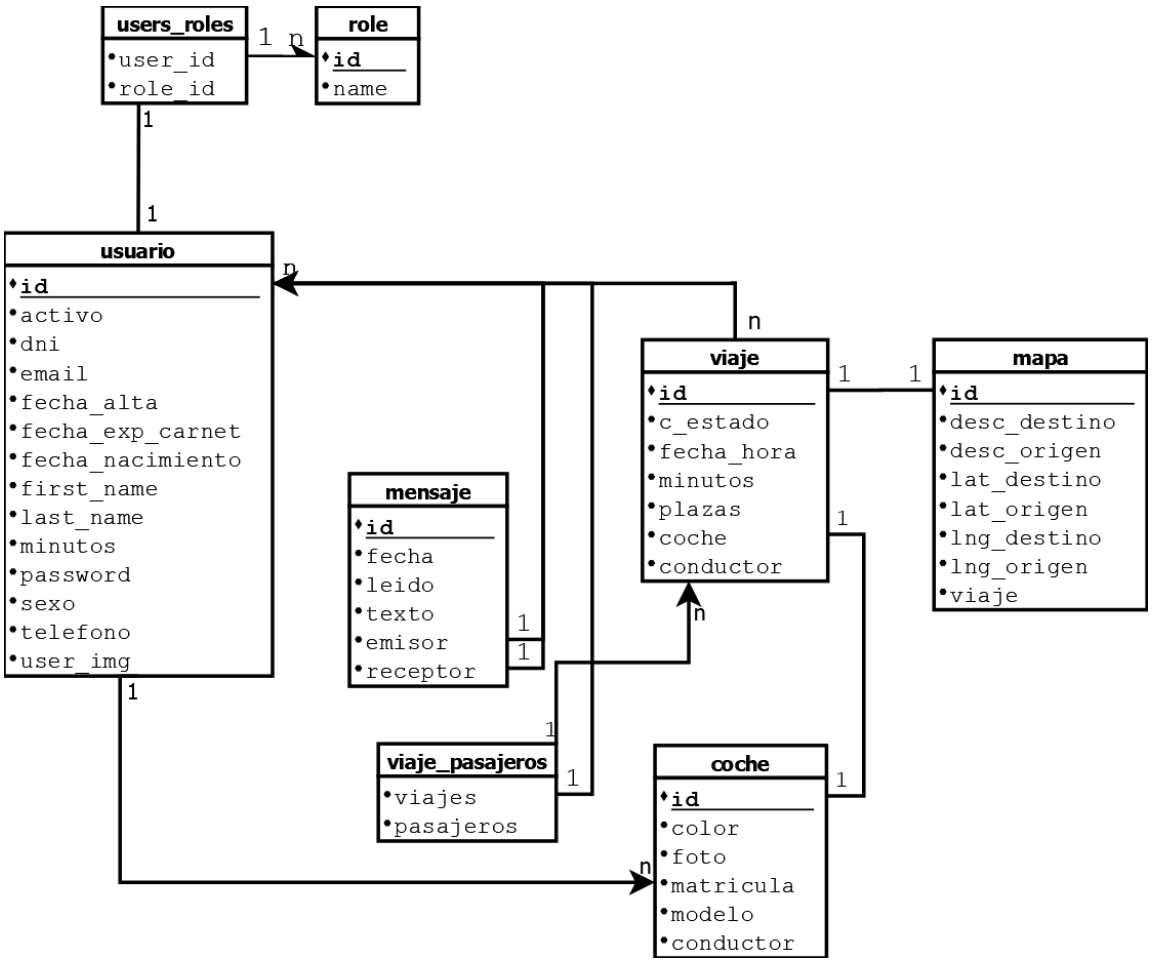

Figura 3.1. Diagrama ER de la base de datos.

3.2 Estructura

En primer lugar, se procederá a explicar la estructura de directorios y organización del código de la aplicación. Así, será más fácil comprender las siguientes secciones.

Como ya se indicó en la sección 2.3, al ser un proyecto Maven, esta aplicación sigue el estándar de directorios especificado por Maven[10]. Se trata de una estructura sencilla que se repite para todos los proyectos Maven, por lo que facilitaría a nuevos desarrolladores acomodarse con el proyecto. En la figura 3.2 se encuentra una captura de la jerarquía de directorios del proyecto.

[10] https://maven.apache.org/guides/introduction/introduction-to-the-standard-directory-layout.html

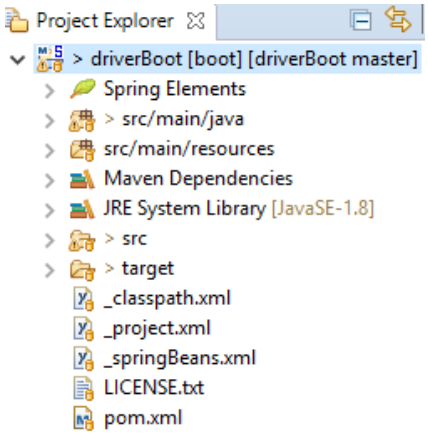

Figura 3.2. Jerarquía de directorios

A continuación, se explican brevemente los directorios más importantes:

- *src/main/java:* en este directorio se encuentra todo el código de Java Spring. Está el paquete general *Driver* en el que se encuentra la clase *Main.java* y el resto de los paquetes:
    - *commons:* se guardan las clases comunes a toda la aplicación, como las constantes, el formulario de retorno y una clase con funciones útiles.
    - *models:* dentro de este paquete están todos los objetos de negocio que representan a las entidades de la base de datos.
    - *inbox:* en este paquete están todas las clases relacionadas con la lógica de la mensajería de la aplicación, como el controlador REST, el DTO (patrón explicado anteriormente en el punto 2.2.1), el *service,* que implementa las reglas de negocio para después interactuar con la base de datos a través de la clase *Repository, etc.*
    - *user:* todas las clases relacionadas con la lógica del usuario, igual que en el punto anterior, incluido también el DTO de coche. Debido a que solo se relacionan con el usuario, no ha sido necesario crear un paquete nuevo.
    - *trip:* todas las clases relacionadas con la lógica del viaje, incluido también el DTO del Mapa. Al igual que con el paquete *user*, los mapas solo se

relacionan con los viajes, por lo que no es necesario crear un paquete nuevo.

- *security:* aquí se encuentran todas las clases necesarias para configurar el módulo de seguridad de Spring.

- *src/main/resources:* se encuentran los recursos necesarios para construir la interfaz *front-end* en el cliente. Aquí también se encuentra el fichero `application.properties` en el que está descrita toda la configuración que debe ser leída por Spring para la conexión a la base de datos y el mapeo con *Hibernate,* explicado anteriormente en la sección 2.2.1.

  Se guarda la carpeta *static* donde irán alojados todos los ficheros necesarios para ejecutar la aplicación en el navegador, i. e.: el fichero `index.html`, la carpeta *js*, donde están todos los ficheros de JavaScript y AngularJS, la carpeta *html*, donde se encuentran todas las vistas HTML, etc.
- *src:* se muestra la jerarquía de carpetas del proyecto, tal y como están guardadas en la memoria local.
- *pom.xml:* es el fichero de configuración propio de *Maven*, en el que se establecen las dependencias del proyecto y sus versiones, como ya se explicó en la sección 2.2.1. En la figura 3.3 se puede ver el contenido del fichero POM de la aplicación. En la etiqueta *parent* (padre), se define el *Framework* con la versión que se utilizará en toda la aplicación. La etiqueta *dependencies*, como su propio nombre indica, contiene todos los módulos de los que depende esta aplicación, como JWT, Spring Security, JPA, etc. Por último, en la etiqueta *properties* se define la codificación, la clase encargada de ejecutar la aplicación y la versión de Java que se va a utilizar.

```xml
<parent>
    <groupId>org.springframework.boot</groupId>
    <artifactId>spring-boot-starter-parent</artifactId>
    <version>1.2.3.RELEASE</version>
    <relativePath />
</parent>

<artifactId>driverBoot</artifactId>
<name>driverBoot</name>

<licenses>
    <license>
        <name>Apache License, Version 2.0</name>
        <url>http://www.apache.org/licenses/LICENSE-2.0</url>
        <distribution>repo</distribution>
    </license>
</licenses>

<dependencies>
    <dependency>
        <groupId>org.springframework.boot</groupId>
        <artifactId>spring-boot-starter-security</artifactId>
    </dependency>
    <dependency>
        <groupId>io.jsonwebtoken</groupId>
        <artifactId>jjwt</artifactId>
        <version>0.7.0</version>
    </dependency>
    <dependency>
        <groupId>org.springframework.boot</groupId>
        <artifactId>spring-boot-starter-web</artifactId>
    </dependency>
    <dependency>
        <groupId>org.springframework.boot</groupId>
        <artifactId>spring-boot-starter-data-jpa</artifactId>
    </dependency>
    <dependency>
        <groupId>commons-beanutils</groupId>
        <artifactId>commons-beanutils</artifactId>
    </dependency>
    <dependency>
        <groupId>mysql</groupId>
        <artifactId>mysql-connector-java</artifactId>
        <scope>runtime</scope>
    </dependency>
</dependencies>

<properties>
    <project.build.sourceEncoding>UTF-8</project.build.sourceEncoding>
    <start-class>driver.Main</start-class>
    <java.version>1.8</java.version>
</properties>
```

Figura 3.3. Fichero POM.xml de la aplicación

3.3 Descripción de los componentes y su interacción

En esta sección se exponen con más detalle los componentes más relevantes desarrollados en esta aplicación, así como la forma en que interactúan entre ellos.

Para ello, se describen los módulos por separado para comentar las interacciones de estos entre cliente y servidor y cómo se ha llevado a cabo la lógica en cada situación.

3.2.1 LOGIN Y SIGN UP

En primer lugar, es necesario entender que gracias al módulo *Security* de Java Spring, solo se cargarán los ficheros JS necesarios para implementar la lógica del *Login y Sign Up*. En la figura 3.4 se muestra la configuración del módulo *security* de la aplicación, en la que se pueden ver los ficheros a los que tienen acceso los usuarios que no están autenticados, las acciones por defecto en caso de haber un *Login* erróneo y la URL por defecto cuando un usuario se registra correctamente o cuando se hace *Logout*.

```
@Override
protected void configure(HttpSecurity http) throws Exception {

    http.csrf().disable().authorizeRequests()
    .antMatchers("/html/addCar.html", "/js/PrincipalCtrl.js", "/registration", "/lib/**",
            "/vendor/**", "/css/**", "/fonts/**", "/images/**").permitAll()
    .anyRequest().authenticated()
    .and()
        .formLogin() // default is /login with an HTTP post
            .loginPage("/login.html")
            .loginProcessingUrl("/login")
            .defaultSuccessUrl("/#/")
            .permitAll()
    .and()
        .logout()
        .invalidateHttpSession(true)
        .logoutRequestMatcher(new AntPathRequestMatcher("/logout"))
        .logoutSuccessUrl("/login.html")
    .permitAll();

}
```

Figura 3.4. Configuración de Spring Security.

Debido a esta razón, el alumno ha decidido que la mejor opción sería crear dos aplicaciones AngularJS, ya que se trata de una página SPA (mencionado en la sección 2.2.1.2). Se ha creado una aplicación *DriverIni*, la cual precisa de un único fichero JS en el Cliente, llamado `Principal.js`. Por otra parte, también se ha creado la aplicación *Driver,* la cual gestiona el grueso de la aplicación y cuenta con el resto de los ficheros JS. En este fichero se encuentra el controlador *PrincipalCtrl* y el controlador de la ventana modal[11] encargada de gestionar el formulario para añadir coches.

La lógica contenida en el controlador *PrincipalCtrl* permite detectar los errores introducidos en el formulario de registro antes de enviar la petición al servidor. Entre otras, cabe destacar que se ha añadido una función para verificar que el DNI es válido (se utiliza un algoritmo para calcular la letra que debe tener), que el usuario es mayor de edad, o que la fecha de expiración del carné de conducir no es superior a la fecha actual. Por supuesto, también se verificará que no haya más de un usuario con el mismo DNI o número de teléfono, el número de teléfono debe pertenecer a un único usuario, ya que sirve de referencia para contactar con dicho usuario.

Además, se ha utilizado la directiva llamada *fileModel,* como ya se ha explicado anteriormente en la sección 2.2.1.2, el uso de directivas permite aplicar acciones sobre el HTML. Esta directiva se crea por la necesidad de introducir imágenes en el formulario y que estas puedan ser leídas desde el controlador, al igual que se utiliza la etiqueta *ng-model* para el resto de los datos. En la figura 3.5 se puede ver el contenido de esta directiva. Por una parte, la opción *restrict* restringe los datos que serán escogidos, en este caso E (lo que permite recoger el nombre de los elementos de la directiva) y A (permite recoger el nombre de los atributos), por defecto suele ser EA. Por otra parte, con la opción *scope*, se define la palabra clave con la que se recogerá el objeto deseado, en este caso la imagen. Por último, la función *link* recoge una serie de parámetros útiles, en este caso solo sirven el *scope* (lo que contiene el *scope* actual de la aplicación) y *ele* (lo que contiene los elementos recogidos con dicha directiva). Dentro de la función *link* se recoge el evento *OnChange*, lo que implica que cuando en esta directiva se introduce

[11] **Ventana Modal:** es un elemento de control gráfico subordinado a la ventana principal de una aplicación. Crea un modo que desactiva la ventana principal, pero la mantiene visible con la ventana modal como una ventana secundaria frente a ella.

un nuevo objeto, se asignará un nuevo valor a la variable del *scope* que contiene la imagen.

Los datos serán convertidos a formato *JSON,* gracias a la función Angular.toJSON(), para pasarlos a través de una llamada POST al servidor.

Para enviar las imágenes al servidor, se ha hecho indispensable utilizar la función *FormData*, la cual se encarga de generar un único objeto en el que se dividen los datos por partes. Por un lado, los datos del formulario y las imágenes y, por otro, los datos de los coches añadidos con sus respectivas imágenes.

```
myAppIni.directive("fileModel",function() {
    return {
        restrict: 'EA',
        scope: {
            setFileData: "&"
        },
        link: function(scope, ele, attrs) {
            ele.on('change', function() {
                scope.$apply(function() {
                    var val = ele[0].files[0];
                    scope.setFileData({ value: val });
                });
            });
        }
    }
})
```

Figura 3.5. Directiva fileModel para leer las imágenes introducidas por el usuario.

Una vez rellenado correctamente el formulario de registro, se realizará la petición POST al servidor. Para leer los datos que llegan en formato JSON se ha utilizado la biblioteca **JSON_MAPPER**. Como se puede ver en la figura 3.6, existe una función llamada *readValue,* la cual percibe dos parámetros: el objeto JSON y la clase a la que se debe mapear el objeto JSON.

```
userDto = Constants.JSON_MAPPER.readValue(userJSON, UserDto.class);
```

Figura 3.6. Uso de la librería JSON_MAPPER para mapear a la clase “UserDto”

Después se asigna valor a todos los atributos del objeto *UserDTO,* el cual actúa como DTO para mover los datos a la clase *UserService*, a través de la función *save*.

En la clase *UserService* se establece valor a todos los atributos de la entidad Usuario, así como también a todos los coches añadidos. A su vez, se utiliza el *bean* creado para encriptar las contraseñas con la clase ***BCryptPasswordEncoder*,** ya explicado en el punto 2.2.1. Para guardar los datos en la base de datos, se ha creado una clase que actúa como repositorio, en este caso es *UserRepository* la cual hereda de *JPARepository*.

Como se puede apreciar en la figura 3.7, será suficiente con llamar a la función *save* para guardar los datos del usuario. Como Coche es atributo de la entidad Usuario y tiene la opción *CascadeType.All*, también se guardarán los coches automáticamente.

```
public Usuario save(UserDto registration){
    Usuario user = new Usuario();
    user.setFirstName(registration.getFirstName());
    user.setLastName(registration.getLastName());
    user.setEmail(registration.getEmail());
    user.setPassword(passwordEncoder.encode(registration.getPassword()));
    user.setUserImg(registration.getUserImg());
    user.setFechaExpCarnet(registration.getfExpiryDate());
    user.setDni(registration.getDni());
    user.setFechaNacimiento(registration.getfBirthDate());
    user.setRoles(Arrays.asList(new Role("ROLE_USER")));
    user.setActivo(true);
    user.setFechaAlta(new Date());
    user.setMinutos(Constants.MINUTOS_REGISTRO);
    user.setTelefono(registration.getPhone());
    user.setSexo(registration.getGender().charAt(0));

    List<Coche> misCoches = new ArrayList<>();
    for(CocheDto coche : registration.getCoches()) {
        Coche miCoche = new Coche();

        miCoche.setColor(coche.getColor());
        miCoche.setConductor(user);
        miCoche.setFoto(coche.getFoto());
        miCoche.setMatricula(coche.getMatricula());
        miCoche.setModelo(coche.getModelo());

        misCoches.add(miCoche);

    }
    user.setCoches(misCoches);

    return userRepository.save(user);
}
```

Figura 3.7. Asignación de los datos provenientes de “UserDto” a la entidad “Usuario” y salvado de los datos a través de userRepository.save

En las figuras 3.8 y 3.9 se muestra la interfaz para ordenador del *Login y Sign Up* respectivamente.

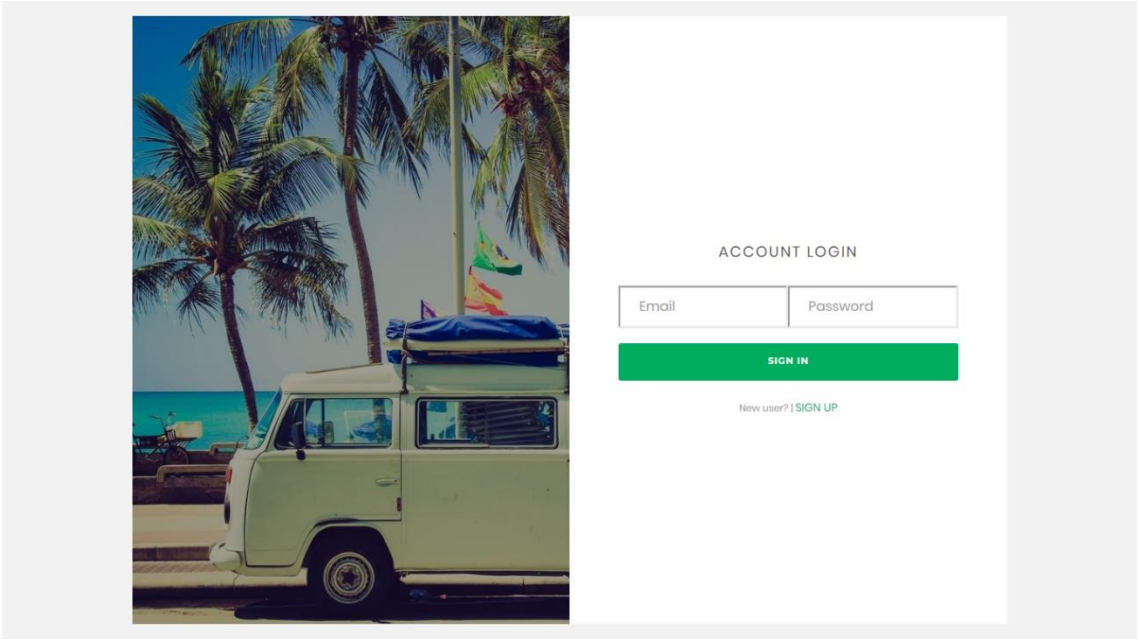

Figura 3.8. Interfaz Login para pantalla ordenador

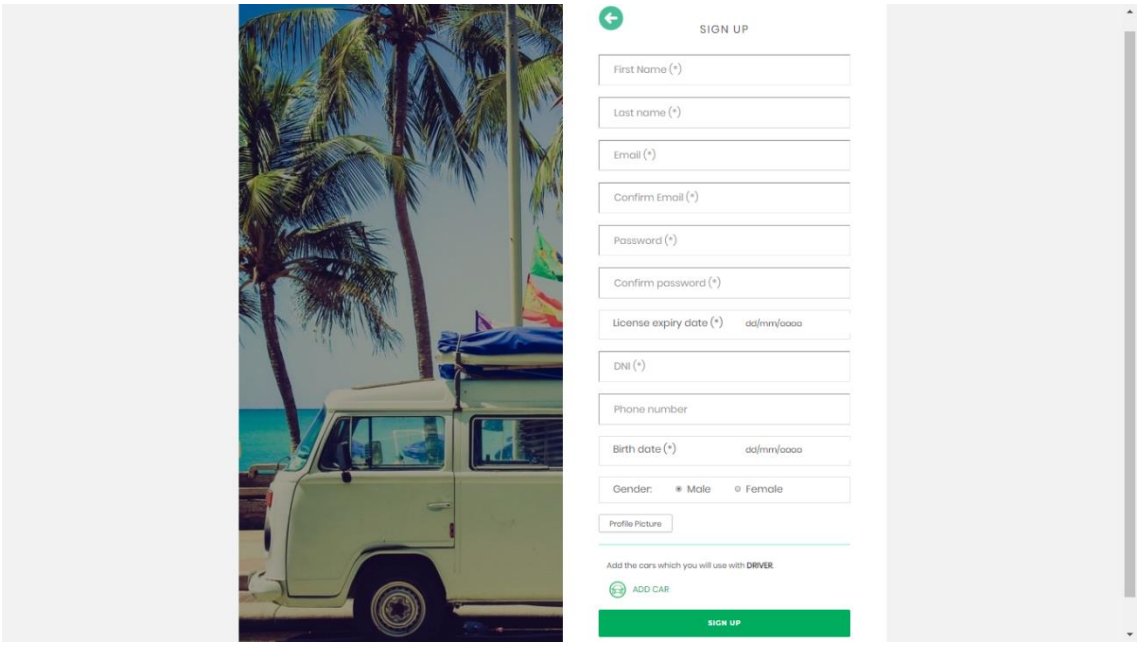

Figura 3.9. Interfaz Sign Up para pantalla ordenador.

3.2.2 ROUTER "APP.JS"

El *router* en AngularJS es uno de los módulos más importantes, ya que permite implementar el concepto SPA.

Para ello, Angular utiliza la directiva *ng-view* en la plantilla HTML, que normalmente suele ser el *index*. El contenedor en el que se encuentra esta directiva es donde se añadirá el código HTML dinámicamente a través de AngularJS.

Se suele destinar un fichero JS para dedicarlo únicamente a implementar el *routing*, en esta aplicación este fichero es `App.js`. Dependiendo de la URL llamada en el navegador, se especificará qué plantilla y qué controlador se debe usar. También se puede especificar qué URL se debe tomar por defecto cuándo el usuario o algún enlace de la aplicación lleve a una URL errónea. Cabe destacar que, al ser este fichero el primero en ejecutarse, también se aprovecha para declarar la variable que pondrá en funcionamiento a AngularJS, esta varible es *myApp*.

En la figura 3.10 se puede ver el contenido del fichero `App.js` de esta aplicación. Cada ruta se especifica en la función *when*, después la plantilla HTML y el controlador que se usarán para dicha ruta, en las etiquetas *templateUrl y controller* respectivamente. También está la función *otherwise*, donde todas las rutas no especificadas anteriormente se redirigirán a "/".

```
var myApp = angular.module('DriverApp',['ngRoute', 'ngAnimate', 'ui.bootstrap']);

myApp.config(function($routeProvider) {
    $routeProvider
    .when("/", {
        templateUrl :"html/map.html",
        controller: "mapCtrl"
    })
    .when("/newTrip", {
        templateUrl :"html/newTrip.html",
        controller: "newTripCtrl"
    })
    .when("/myTrips", {
        templateUrl :"html/myTrips.html",
        controller: "myTripsCtrl"
    })
    .when("/inbox", {
        templateUrl :"html/inbox.html",
        controller: "inboxCtrl"
    })
    .when("/myAccount", {
        templateUrl :"html/myAccount.html",
        controller: "myAccountCtrl"
    })
    .otherwise({redirectTo:'/'});

});
```

Figura 3.10. Fichero “App.js” router de AngularJS.

3.2.3 MAP

Es la ventana principal que se verá una vez que se haya iniciado sesión. En esta ventana se muestra un mapa que ocupa prácticamente toda la pantalla donde se pueden ver todos los viajes publicados, siempre y cuando la fecha/hora de estos viajes sea superior a la actual. Además, también se muestra un campo de texto para filtrar la búsqueda de viajes por fecha. Dentro del mapa, se ha incluido otro campo de texto para filtrar los viajes por localización que, por defecto, será la localización donde se encuentre el usuario en caso de tener permiso de geolocalización. En caso contrario, se abrirá por defecto el mapa de Madrid.

Para hacer esto posible se ha creado una *Factory* de Angular llamada *MantenimientoSrv*, la cual nos permite declarar funciones que se pueden usar a lo largo de la aplicación. Es similar a una clase estática de Java. Esta factoría está dedicada a realizar peticiones web,

que pueden devolver un valor satisfactorio o un error. Esto se ha hecho para facilitar la lectura y la organización del código.

La petición web será recogida por el controlador REST del servidor Java Spring, que recogerá los datos necesarios, los procesará y retornará al cliente la información requerida. A continuación, se explicará un ejemplo de esta llamada desde el cliente al servidor, que funcionará de forma prácticamente igual en el resto de las peticiones.

3.2.3.1 Peticiones Web

En primer lugar, se realizará la llamada *getViajes()* a *MantenimientoSrv* como se muestra en la figura 3.11.

```
function getViajes(){
    var deferred = $q.defer();
    var promise = deferred.promise;

    function success(data){
        var viajes = data.data;

        viajes.forEach(function(viaje){
            viaje.fechaHora = new Date(viaje.fechaHora);
        });


        deferred.resolve(viajes);
    };

    function error(data){

        deferred.reject(data);
    };


    $http({
            url: '/getTrips',
            method: 'GET',
            headers: {'Content-Type': undefined},
            transformRequest: angular.identity
        }).then(success , error);

    return promise;

}
```

Figura 3.11. Función getViajes() en MantenimientoSrv.

Después de esta llamada, la petición será recibida en el controlador REST de Java, el cual llamará al *Service* encargado de implementar las reglas de negocio y devolver el objeto pedido. En la figura 3.12 se puede ver la función que recibe la petición en el controlador REST.

```
@Autowired
private ViajeService viajeService;

@RequestMapping(value="/getTrips",method = RequestMethod.GET)
@ResponseBody
public List<ViajeDto> getTrips(final HttpServletRequest request, HttpServletResponse response) {
    List<ViajeDto> misViajes = new ArrayList<ViajeDto>();

    misViajes = (List<ViajeDto>) viajeService.getViajes();

    return misViajes;

}
```

Figura 3.12. Función getTrips del controlador REST

3.2.3.2 InfoWindow y Markers

Los ***Markers*** son iconos mediante los cuales se pueden señalar puntos en el mapa, en este caso sirven para saber dónde se encuentran los viajes disponibles. Cuando se pulsa sobre uno de los marcadores, se abrirá una ventana dentro del mismo mapa que contendrá información sobre el *marker*, a esta ventana se le llama ***InfoWindow***.

En esta aplicación, además de mostrar la información sobre el viaje en la ventana *InfoWindow*, también se mostrará automáticamente la ruta que tomará el viaje, así también se mostrará el lugar de destino. En la figura 3.13 se muestra un ejemplo.

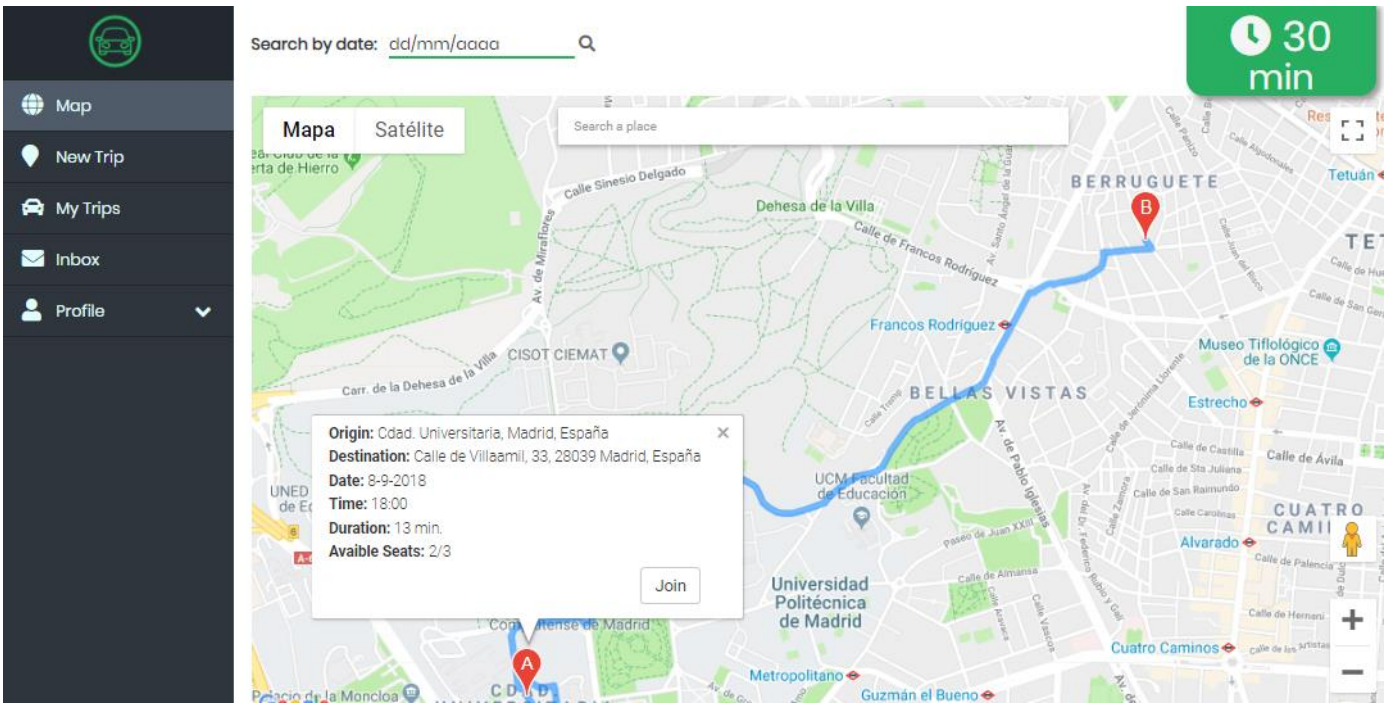

Figura 3.13. Ventana InfoWindow y ruta.

La ventana *InfoWindow* ha tenido especial complicación, ya que el código HTML que contiene no ejecuta AngularJS por defecto. Para acoplar esto con la aplicación AngularJS se ha tenido que usar la biblioteca *compile,* la cual ofrece la funcionalidad de compilar el código HTML antes de mostrarse. Así, ha sido posible añadir botones y la información necesaria a través del *scope* como se venía haciendo hasta ahora. En la figura 3.14 se muestra el código que ha sido necesario para implementar esta función.

```
$scope.viajes.forEach(function(viaje){

  var horaStr = utils.horaToStr(viaje.fechaHora);
  var fehaStr = utils.fechaToStr(viaje.fechaHora);
  var contentString =
  '<div> <label>Origin:  </label><span> '+viaje.mapa.descOrigen + '</span>' +
  '<br/> <label>Destination:  </label><span> '+viaje.mapa.descDestino +'</span>' +
  '<br/> <label>Date:  </label><span> '+ fehaStr + '</span>' +
  '<br/> <label>Time:  </label><span> '+ horaStr + '</span>' +
  '<br/> <label>Duration:  </label><span> '+viaje.minutos + ' min.</span>' +
  '<br/> <label>Avaible Seats:  </label><span> '+ (viaje.plazas - viaje.pasajeros.length).toString() +'/'+ (viaje.plazas).toString() +'</span>' +
  '<br/> <button style="float: right;" type="buttton" class="btn btn-default" ng-click="showModal('+ viaje.id +');">Join</button></div>';
  var compiledContent = $compile(contentString)($scope);

  var infowindow = new google.maps.InfoWindow;
  infowindows.push(infowindow);
```

Figura 3.14. Uso de la herramienta $compile de AngularJS para añadir información a InfoWindow.

Pulsando sobre el botón *Join* de la ventana *InfoWindow*, se abrirá una ventana modal con todos los detalles del viaje, pasajeros y coche, incluyendo la posibilidad de mandar un mensaje al conductor o pasajeros, y por supuesto la posibilidad de unirse al viaje.

Para ello, se comprueba que haya sitio disponible y el usuario tenga minutos suficientes para "pagar" el viaje, en caso de que no se cumpla, se mostrará el error en la parte superior de la ventana. En la figura 3.15 se muestra una imagen donde se muestra esta ventana modal.

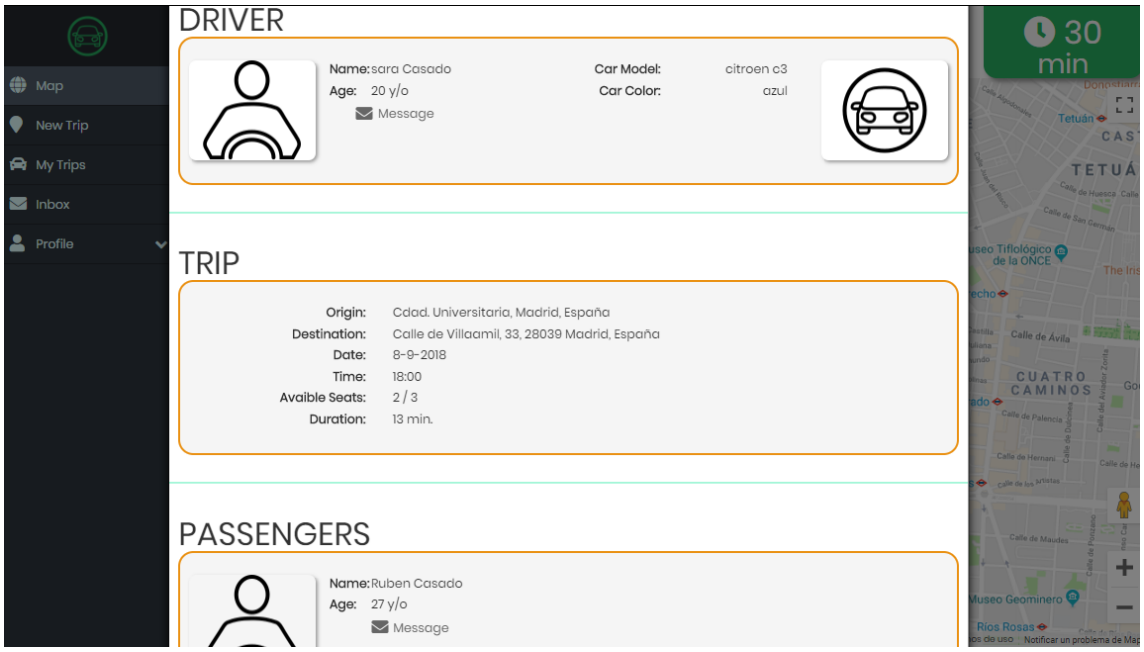

Figura 3.15. Ventana modal para unirse a un viaje.

Es importante destacar que cuando un usuario se une a un viaje se le restarán los minutos correspondientes, entonces, para actualizar la etiqueta que muestra los minutos (parte superior de la derecha) es necesario mandar un evento al controlador padre "MenuCtrl" con *$scope.$emit* y se recoge este evento con la función *$scope.$on.* En las figuras 3.16 y 3.17 se muestra el código necesario para esta implementación, la clase MapCtrl que lanza el evento y MenuCtrl que lo recoge.

```
//enviar evento para cambiar minutos
$scope.$emit('minutos', $scope.usuario.minutos);
```

Figura 3.16. Clase MapCtrl que envía el evento a la clase.

```
//recoger evento si se cambian minutos (al hacer join)
$scope.$on('minutos', function(evt, min){
    $scope.usuario.minutos = min;
});
```

Figura 3.17. Clase MenuCtrl, clase padre que recibe el evento y actualiza la variable que contiene los minutos.

3.2.4 NEW TRIP

La funcionalidad *New Trip* permite al usuario crear y compartir nuevos viajes con el resto de usuarios. Para ello, se rellenará un pequeño formulario con los datos necesarios para el viaje, como el origen y destino, la fecha y hora de salida, el coche que se va a utilizar y el número de asientos disponibles.

El campo origen y destino cuentan con la función *autocomplete* ofrecida por la biblioteca *places* de la API de Google Maps. Esta función permite identificar si la información sobre el origen o destino es correcta y, en caso de serlo, se utiliza la función *route* de *DirectionsService*. Esta ruta será mostrada en un pequeño mapa que se encuentra debajo del formulario, además, también se mostrará el tiempo estimado que durará el viaje. En la figura 3.18 se puede ver un ejemplo.

Una vez que se completa el formulario, y tras pasar las comprobaciones necesarias de los datos, se podrá añadir el viaje a la base de datos y así ser mostrada en el mapa de la aplicación.

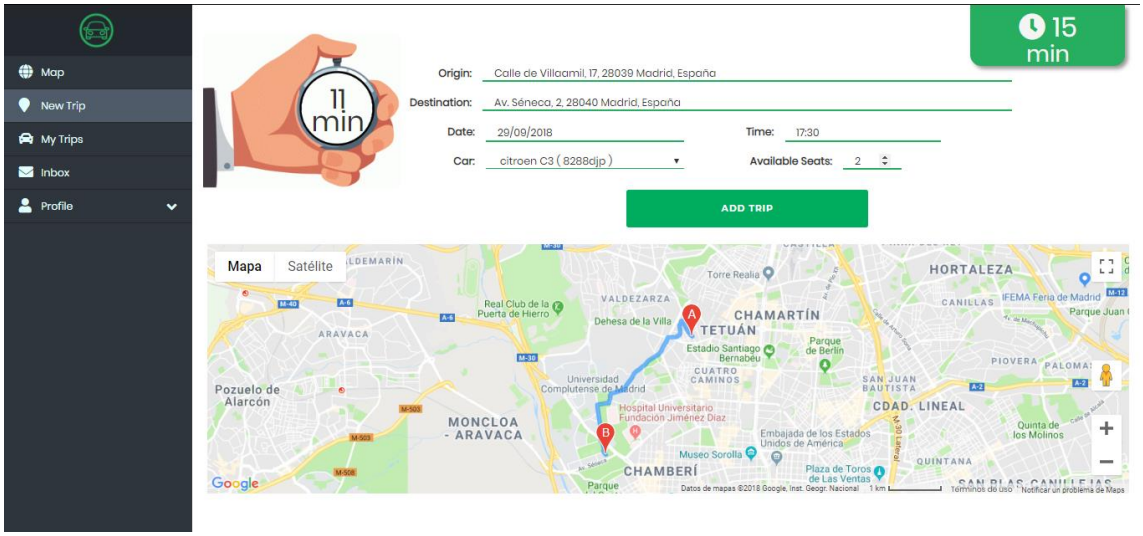

Figura 3.18. Interfaz de la funcionalidad “New Trip”

3.2.5 MY TRIPS

Esta funcionalidad de la aplicación permitirá al usuario poder visualizar todos sus viajes, además, como se puede ver en la figura 3.19, incluye unas pestañas a través de las cuales se podrá filtrar el contenido por el rol que toma el usuario en cada viaje (Conductor/Pasajero).

También se incluye un botón en la columna *Details* que permite abrir una ventana modal con todos los detalles del viaje y desde esta se podrá enviar un mensaje al conductor o pasajeros y ver el teléfono móvil del conductor por si fuera necesario. Esta modal será prácticamente igual a la figura 3.15, a diferencia de que en esta no aparecerá el boton *Join* y sí aparecerá el teléfono del conductor.

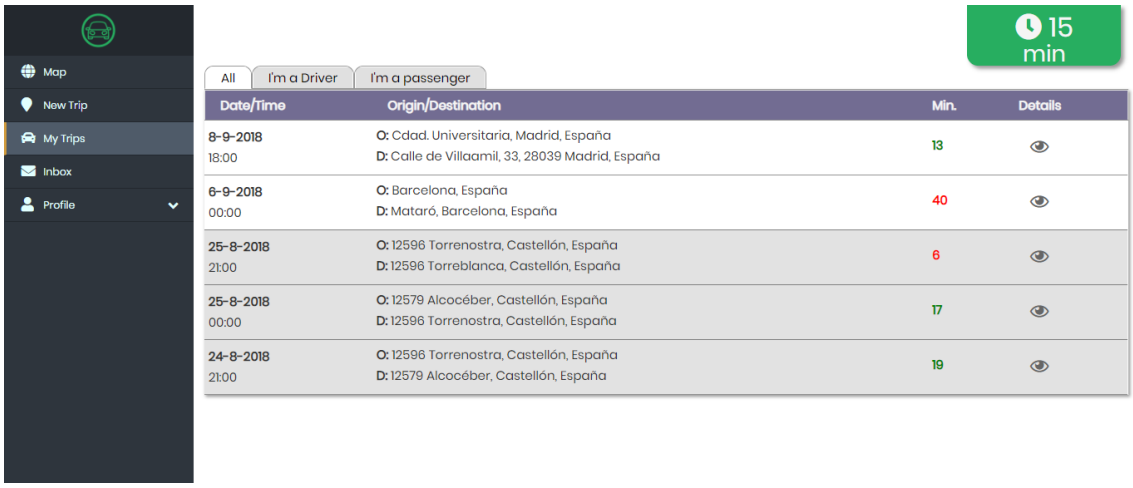

Figura 3.19. Interfaz “My Trips”.

3.2.6 INBOX

Para esta aplicación, al tratarse de una constante interacción entre usuarios, ha sido necesaria la creación de una plataforma de mensajería. Esto permite a los usuarios comunicarse a través de la aplicación.

Para explicar el desarrollo de esta funcionalidad es importante recordar que, en la entidad Usuario, existen dos atributos para los mensajes, por un lado, los mensajes

enviados y, por otro, los recibidos. Esto facilita bastante las cosas a la hora de la lectura de los mensajes por parte del cliente.

Para mostrar todas las conversaciones del usuario ordenadas por el receptor y por la fecha y hora del último mensaje, como se muestra en la figura 3.20, ha sido necesario crear un *HashMap<Key,Value>*[12], teniendo como *key* el *email* del receptor, y como *value* todos los mensajes con ese usuario.

Para ello, se llama a la función "mensajesToMap()" donde se recorren todos los mensajes del usuario, tanto recibidos como enviados, y se introducen en el *HashMap*. Una vez formado el mapa, se procede a llamar a la función "getDatosMuestra()", la cual se encarga de formar una cadena de objetos ya ordenados con los datos que se mostrarán en la ventana del *inbox*, como la fecha/hora, nombre del receptor y último mensaje (ver figura 3.20).

Además, también se marcarán las conversaciones con mensajes pendientes por leer y al lado de la etiqueta de "*Inbox*" del menú, aparecerá una marca indicando que hay mensajes sin leer. Para ello, cada vez que se carguen los datos del usuario, se comprobará si existen nuevos mensajes, emitiendo un evento como se explicó anteriormente (similar a las figuras 3.16 y 3.17). Un ejemplo de ello se puede ver en la figura 3.20.

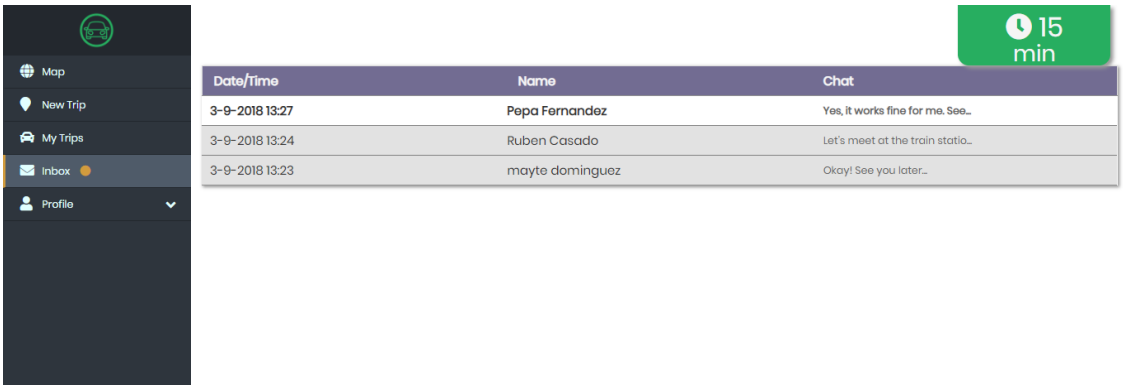

Figura 3.20. Interfaz de la ventana "Inbox" con mensajes pendientes de leer.

[12] **HashMap:** es una clase de contenedor asociativo para almacenar pares clave y valor, siendo esta clave única en el mapa.

3.2.6.1 Ventana Chat

Si se pulsa sobre cualquiera de las conversaciones, se abrirá una ventana modal con todos los mensajes entre el usuario y el receptor ordenados por fecha.

Además, también es posible saber si el receptor ha leído los mensajes, ya que aparecerá un *tick* cuando esto suceda. Se puede ver un ejemplo en la figura 3.21, donde el último mensaje enviado no ha sido leído, pero los anteriores sí.

Para conseguir esta funcionalidad, cada vez que se abre un *chat* se hace una petición al servidor, mandando como parámetro los datos del receptor, y se actualiza el atributo "leído" de todos los mensajes recibidos por ese receptor.

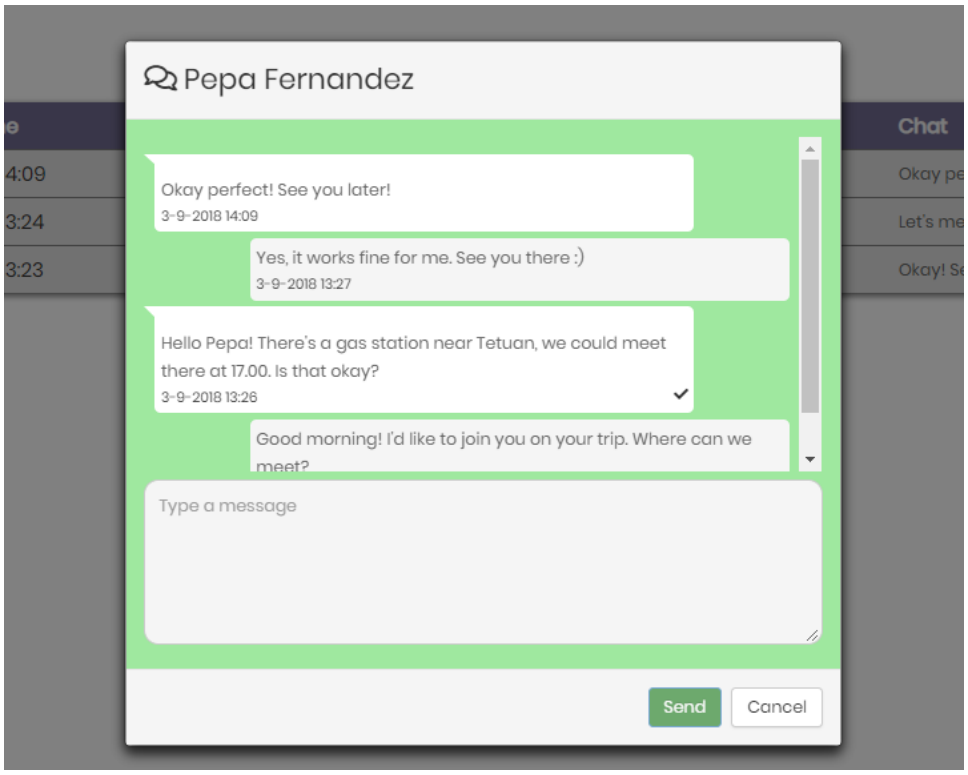

Figura 3.21. Ventana Modal del Chat. El último mensaje enviado no ha sido leído.

3.2.7 MY ACCOUNT

Por último, también existe "*My Account*" que, como se muestra en la figura 3.22, sirve para mostrar toda la información personal del usuario, así como de sus coches añadidos.

Es una funcionalidad sencilla, ya que simplemente se muestran los datos del objeto "$scope.usuario".

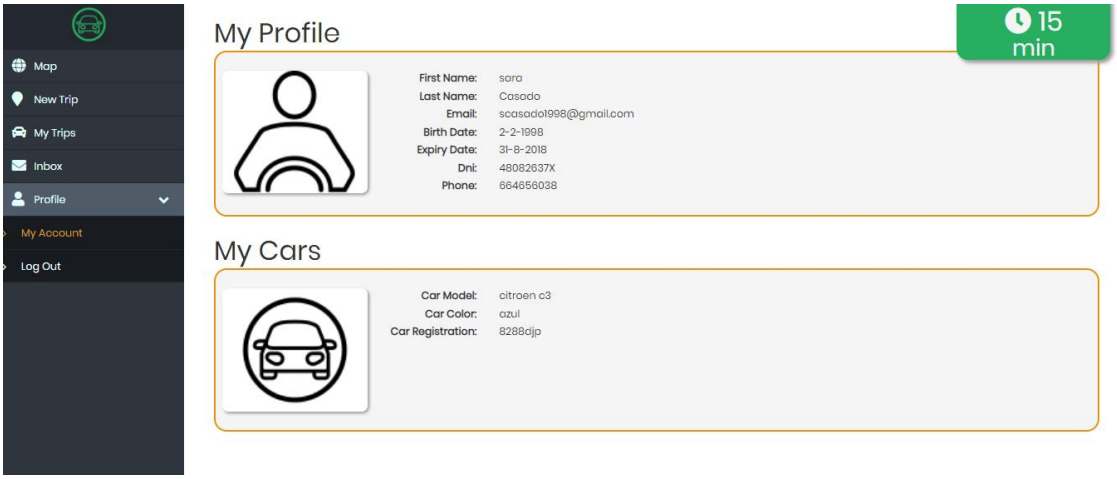

Figura 3.22. Ventana "My Account".

4. CONCLUSIÓN Y TRABAJO FUTURO

Esta sección se divide en dos apartados, por un lado, se dará una opinión y crítica personal del alumno hacia el proyecto realizado y, por otro lado, se expondrán ideas útiles que se podrían utilizar para una futura ampliación y mejora de la aplicación.

4.1 Valoración crítica

En primer lugar, cabe destacar que el proyecto se realizó con un tiempo muy limitado, ya que no se pudo comenzar con el desarrollo del código hasta abril. Este hecho ha limitado las funcionalidades que tenía previstas en un primer momento. Otro problema que añadir es la poca o nula experiencia con algunas de las tecnologías utilizadas, lo que conllevó un aumento del tiempo requerido, debido a la curva de aprendizaje.

Por otra parte, me encuentro orgulloso y satisfecho con el trabajo realizado, ya que a pesar de haber tenido contratiempos y haber tenido que dejar atrás algunas funcionalidades, pude implementar lo más complejo e importante, como el correcto funcionamiento de mapas, creación de nuevos viajes, visualización de rutas, cálculo del tiempo de cada ruta y plataforma de mensajería. Algo que me motivó fue el tratarse de un proyecto basado en una idea personal. Aunque siempre se puede seguir depurando, creo que los errores más relevantes están solucionados, incluso he podido permitirme el lujo de incluir algunos pequeños detalles en estilos y funcionalidades.

Finalmente, me gustaría recalcar que con este proyecto he aprendido más de lo esperado en el diseño e implementación de aplicaciones web, tanto la parte del servidor como del cliente, con algunos de los *frameworks* más importantes del momento. Algo que estoy totalmente seguro me ayudará en el futuro, tanto laboral como personalmente.

4.2 Trabajo futuro

Este proyecto, como ya se ha comentado en el punto anterior, requiere de muchas más funcionalidades para realmente servir a su propósito. Además de perfeccionar el trabajo ya realizado en este proyecto.

Algunas de las posibles funcionalidades que se podrían añadir son, por ejemplo, realizar verificaciones por email y SMS de las cuentas creadas para una mejora de la seguridad. Además, creo que sería imprescindible verificar el carné de conducir en una web oficial, ya que de lo contrario se podría poner en peligro la seguridad de los usuarios, además de infringir la ley.

También sería bastante útil poder añadir una valoración de los usuarios con los que se ha viajado, lo que seguro añadiría confianza para futuros viajes. Así como poder añadir una reclamación en caso de no presentarse el conductor o los pasajeros, y que esta reclamación pueda ser revisada por los administradores. Para ello, también sería necesario añadir la funcionalidad necesaria para desvincularse o eliminar un viaje, y avisar al conductor o pasajeros (según corresponda) de los cambios realizados.

Estas y muchas más ampliaciones se podrían dar para mejorar esta herramienta que, sin duda, mejoraría el concepto de transporte y economía que existe actualmente en la sociedad.

4. CONCLUSIONS AND FUTURE WORK

On this section, two main points will be explained. First, a critical analysis and thoughts about the making of this project. Also, ideas that could be useful to do a future extensión and improvement of the application.

4.1 Critical Analysis

First of all, it should be mentioned that time was tight, since it wasn't possible to start developing until April. This fact restricted the planned features. Another setback was the non-existent experience on the used technologies, which rose up the amount of required time, due to the learning curve.

Notwithstanding, i found myself proud and satisfied with my work. Despite of the mishaps and having to eliminate some features of my plan, i could implement the most complex and important things, such as maps, adding trips, trip display, time estimation and the message box. Part of my motivation lies in the fact that this project is based on a personal idea. Although it can always be debugged, the more relevant errors have been resolved. I even had some time to include small details in style and features.

Finally, I want to highlight that I have learned much more than expected on design and implemention of web applications, both for the server and for the client, using some trending frameworks. I believe all this experience will be useful for me in the future.

4.2 Future work

As explained above, this project requires much more features and improvements to really achieve its purpose.

Some of the additional functionalities would be e-mail and SMS verification to improve security. Moreover, it would be essential to verify the driving license, since user's life could be endangered if someone faked it.

Another idea is the possibility of posting feed-back of users (drivers and passengers), it would boost confidence to future trips. As well as filing a complaint when the driver or passenger doesn't show up; this report would be reviewed by managers. It would also be necessary the feature to withdraw or eliminate a trip, as well as warning the users engaged.

All the extensions mentioned above and much more might be done in the future in order to improve this tool, which Will for sure change the current concept of personal transport and economy.

BIBLIOGRAFÍA

1. *Crowd-Sourced Carpool Recommendation Based on.* **DongWoo, Lee y Liang, Steve.** Calgary, Canada : Proceedings of the 4th ACM SIGSPATIAL International Workshop on Computational Transportation Science, 2011.

2. **Collom, Ed, Lasker, Judith y Kyriacou, Corinne.** *Equal Time, Equal Value: Community Currencies and Time Banking in the US.* Burlington, USA : Ashgate, 2012.

3. **Sonatype Company.** *Maven: The Definitive Guide.* s.l. : O'Reilly, 2008.

4. **Varanasi, Balaji y Belida, Sudha.** *Spring REST.* Berkley : Apress, 2015.

5. **Winch, Robert y Mularien, Peter.** *Spring Security 3.1.* Birmingham : Packt Pub, 2012.

6. **Keith, Mike y Schincariol, Merrik.** *Pro JPA 2.* Berkley : Apress, 2013.

7. **Camps, Rafael, y otros.** *Bases de datos.* Barcelona : Eureca Media SL, 2005.

8. **Jiménez Capel, María Yolanda.** *Bases de datos relacionales y modelado de datos.* s.l. : IC Editorial, 2014.

9. *A Relational Model of data for Large Shared Data Banks.* **Codd, Edgar Frank.** 6, San Jose, California : Communications of ACM, 1970, Vol. 13.

10. **Alur, Deepak, Crupi, John y Malks, Dan.** *Core J2EE patterns : best practices and design strategies.* New Jersey : Prentice Hall, 2007.

11. **Flower, Martin.** *Patterns of Enterprise Application Architecture.* s.l. : Addison-Wesley, 2002.

12. **Spurlock, Jake.** *Bootstrap.* s.l. : O'Reilly Media, 2013.

13. **Goodman, Danny.** *JavaScript bible.* Hoboken : Wiley, 2010.

14. **Green, Brad y Seshadri, Shyam.** *AngularJS.* Sebastopol : O'Reilly Media, 2013.

15. **Fielding, Roy Thomas.** Architectural Styles and the Design of Network-based Software Architectures. [En línea] 2000.

16. **Google Developers. [En línea] https://developers.google.com/.**

ANEXO I. GUÍA DE INSTALACIÓN EN WINDOWS 10

A continuación, se mostrará una breve guía sobre la instalación de la herramienta. Esta guía ha sido elaborada en un equipo con sistema operativo Windows 10, por lo que no se puede asegurar que funcione de la misma manera en el resto de los sistemas operativos.

1. Descargar e instalar el IDE Eclipse desde el siguiente link: https://www.eclipse.org/downloads/ . Para esta guía se ha utilizado la versión Eclipse Oxygen.1a Release (4.7.1a).
2. Descargar e instalar la herramienta XAMPP del link: https://www.apachefriends.org/es/download.html . En esta guía se ha utilizado la versión XAMPP Control Panel v3.2.2.
3. Descargar el código fuente de esta aplicación del siguiente link: https://github.com/RubenK01/driverBoot .
4. Importar el código fuente al IDE Eclipse. Importar como "proyecto Maven" desde la pestaña `File → Import → Maven → Existing Maven Projects`, en la figura A.1 se puede ver un ejemplo.

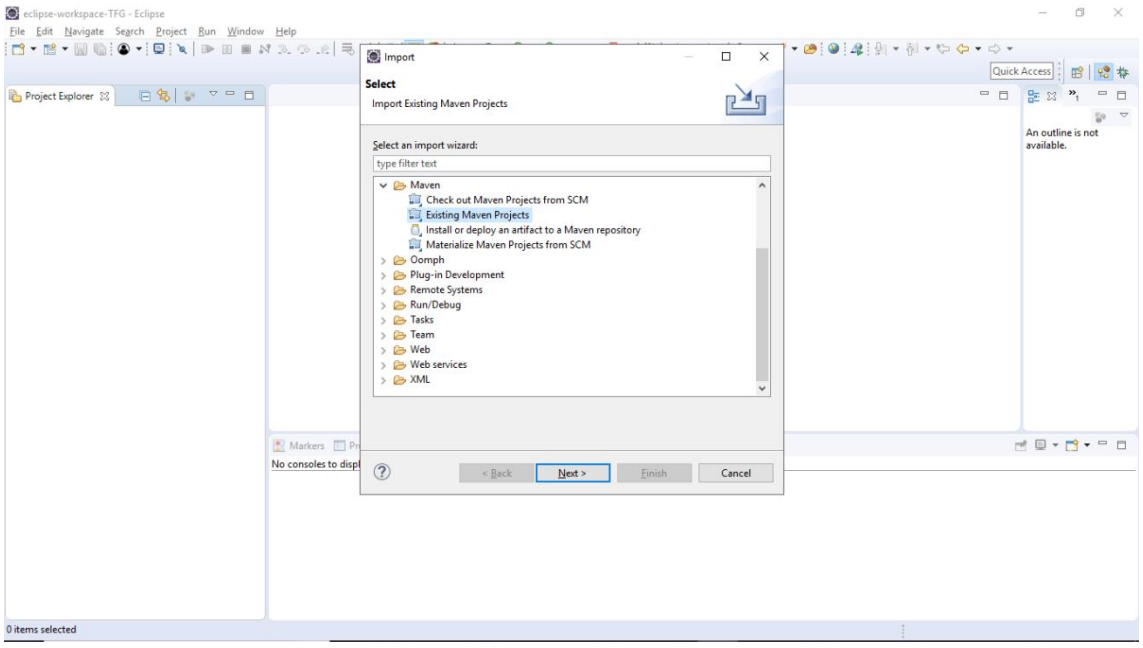

Figura A.1. Importar Proyecto Maven de proyecto existente

5. En la siguiente ventana, pulsar `Browse`, seleccionar la carpeta que contiene el código fuente y pulsar `Aceptar`
6. Asegurarse de que el fichero `pom.xml` está marcado, como aparece en la figura A.2. Pulsar `Finish`.

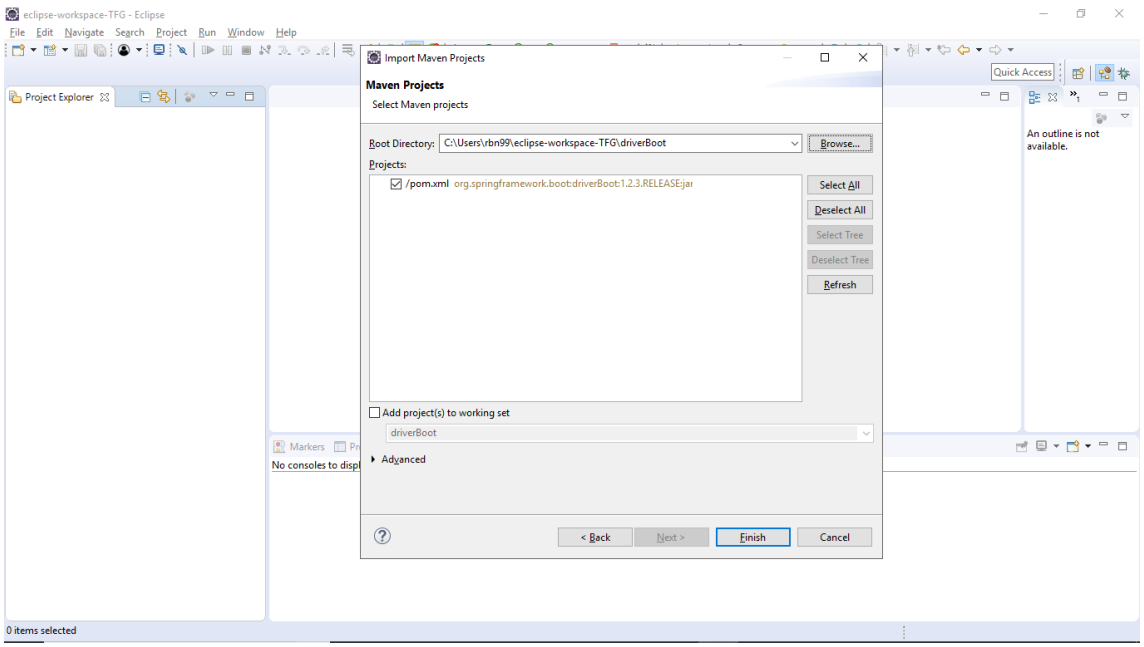

Figura A.2. Selección del proyecto Maven.

7. Abrir XAMPP e iniciar los módulos Apache y MySQL pulsando sobre `start`. Ver figura A.3.

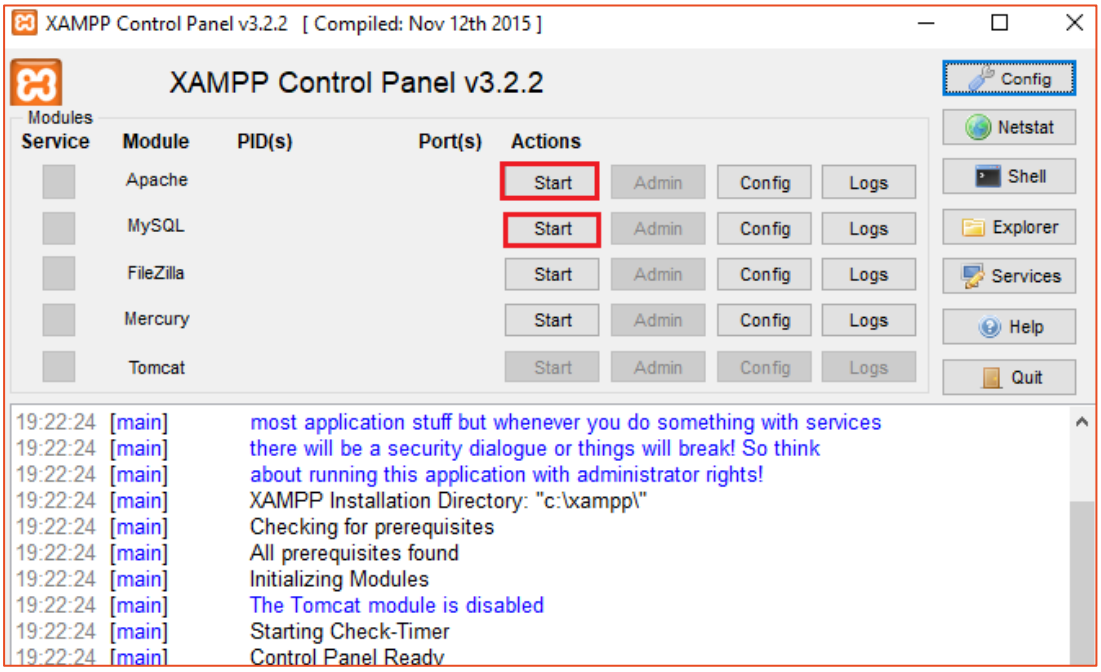

Figura A.3. Inicio de módulos Apache y MySQL en XAMPP

8. Abrir el navegador. Ir a http://localhost/phpmyadmin/ . Ir a la pestaña `Bases de datos`. En el campo `Nombre de la base de datos` introducir "driver" y en cotejamiento, seleccionar `utf8_spanish_ci`, y pulsa `crear`. Ver figura A.4.

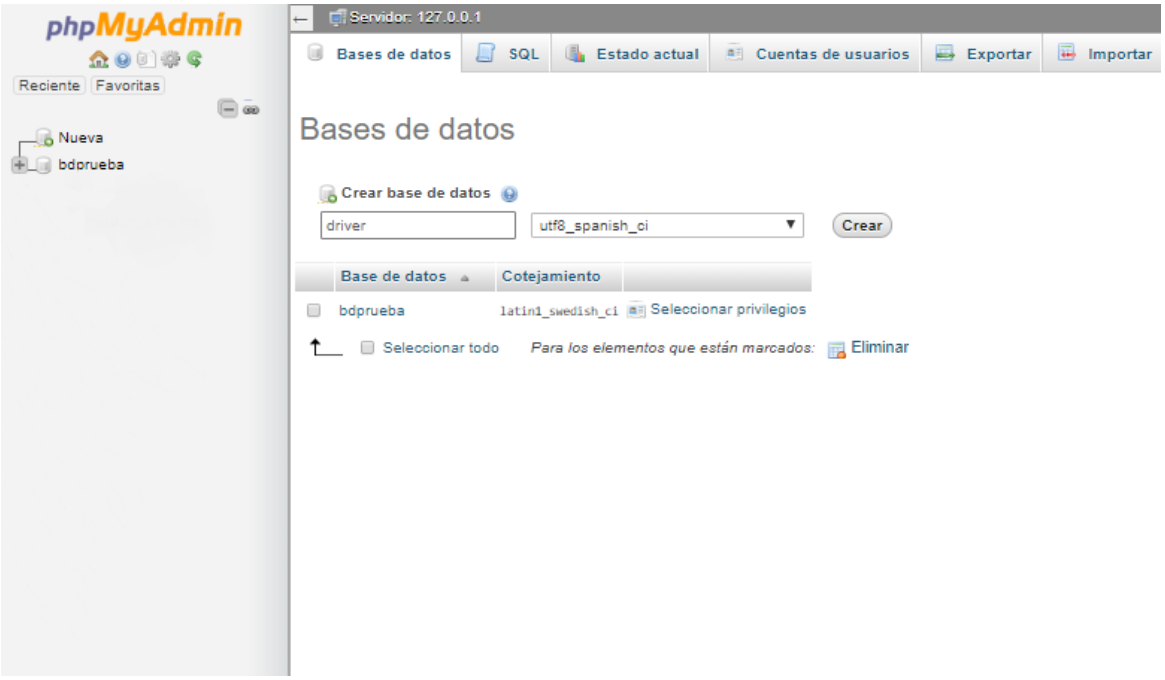

Figura A.4. Creación de base de datos "driver"

9. Ir al proyecto que se ha importado en el paso 6. Ir a `src` → `main` → `resources` → `application.properties`. En la línea 33, cambiar `update` por `create-drop`, como se muestra en la figura A.5.

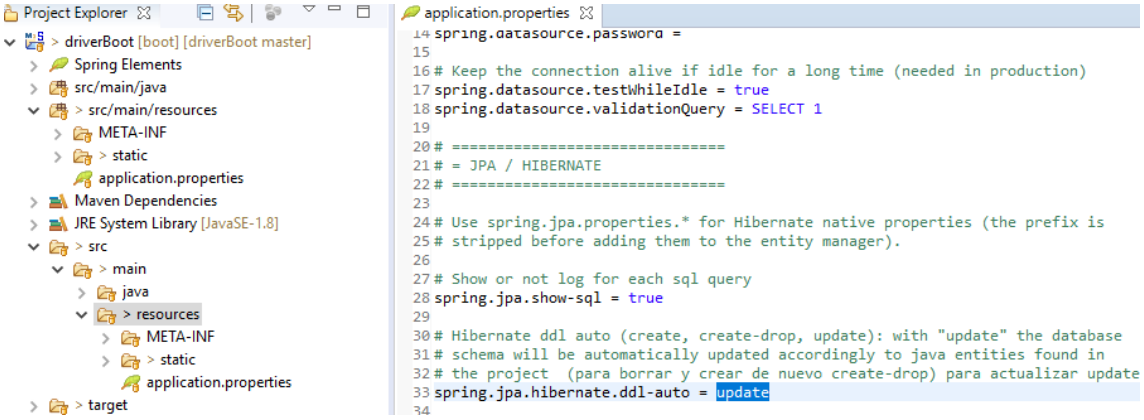

Figura A.5. Configuración "application.properties" inicio base de datos

10. Ir a `src` → `main` → `java` → `driver`. Seleccionar el fichero `Main.java`. En la barra de herramientas, seleccionar `Run` → `Run As` → `JavaApplication`. Ver figura A.6.

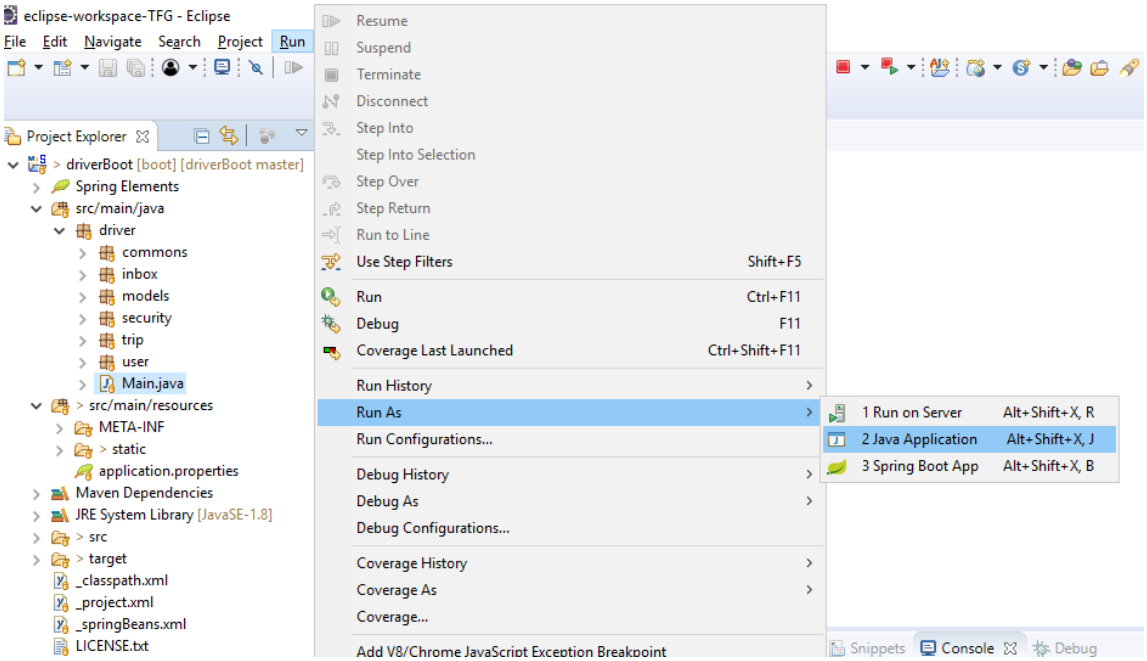

Figura A.6. Ejecutar la aplicación.

11. Abrir el navegador e ir a http://localhost:8080/login.html#/ . Aquí ya se puede ver la aplicación.
12. Por último, hacer el paso 9 a la inversa: cambiar `create-drop` por `update`.

ANEXO II. INTERFACES EN TAMAÑO MOVIL/TABLET

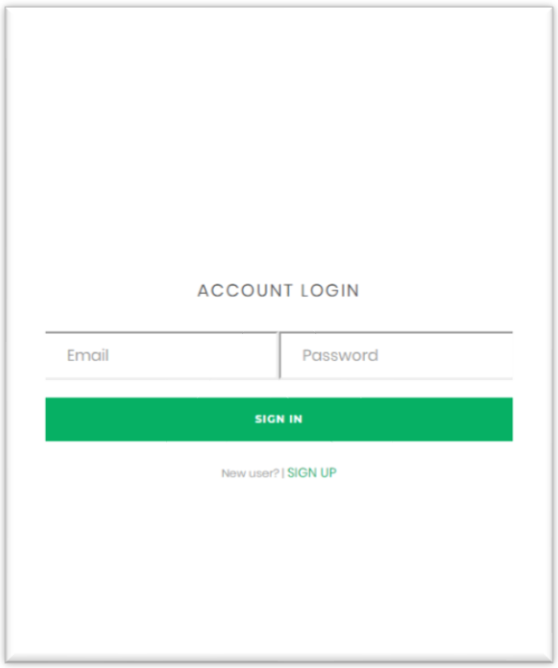

Figura A.7. Interfaz Login.

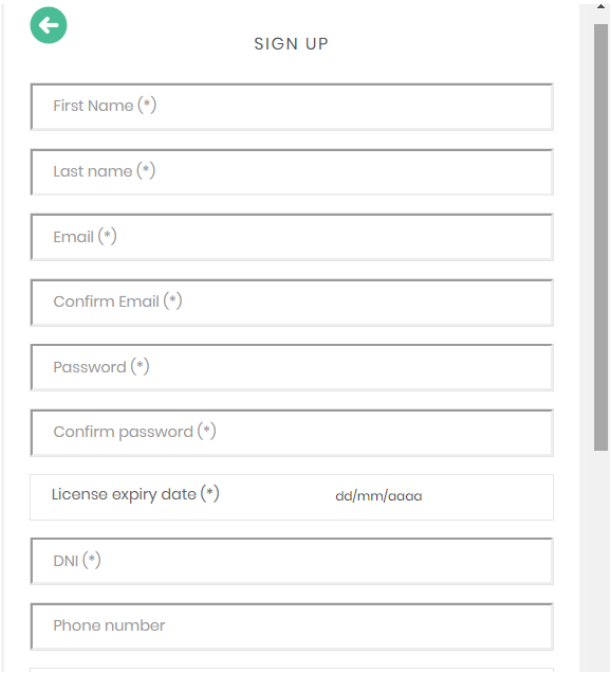

Figura A.8. Interfaz Sign Up

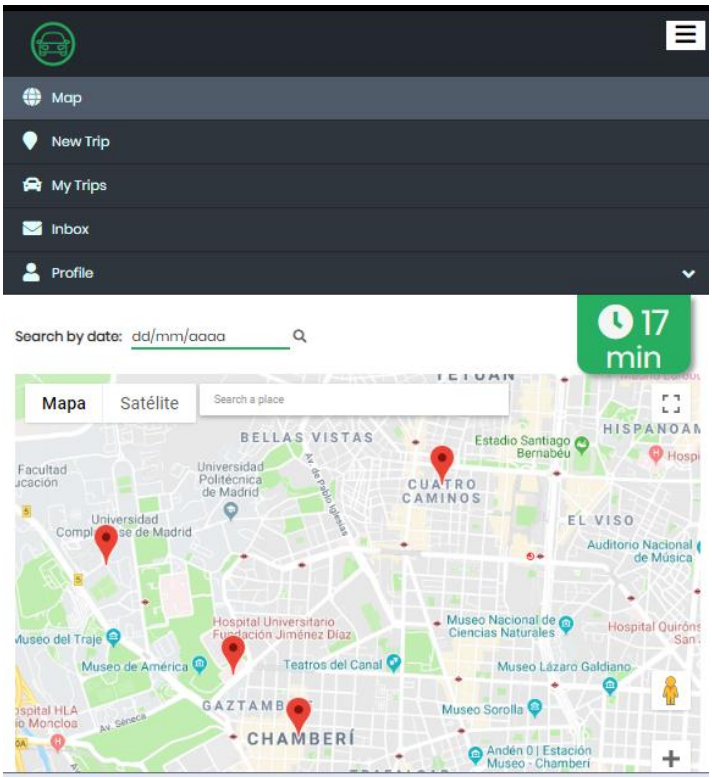

Figura A.9. Interfaz Map con menú abierto.

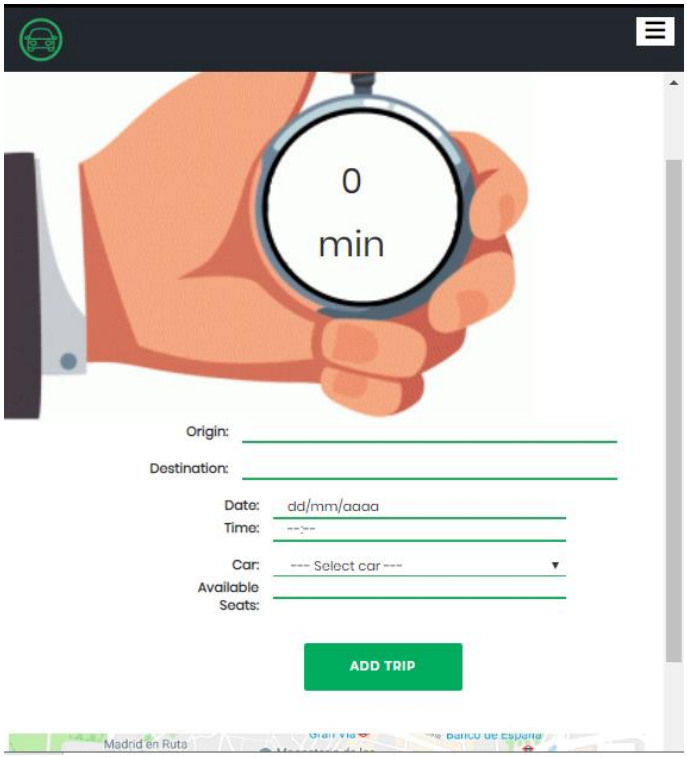

Figura A.10. Interfaz New Trip

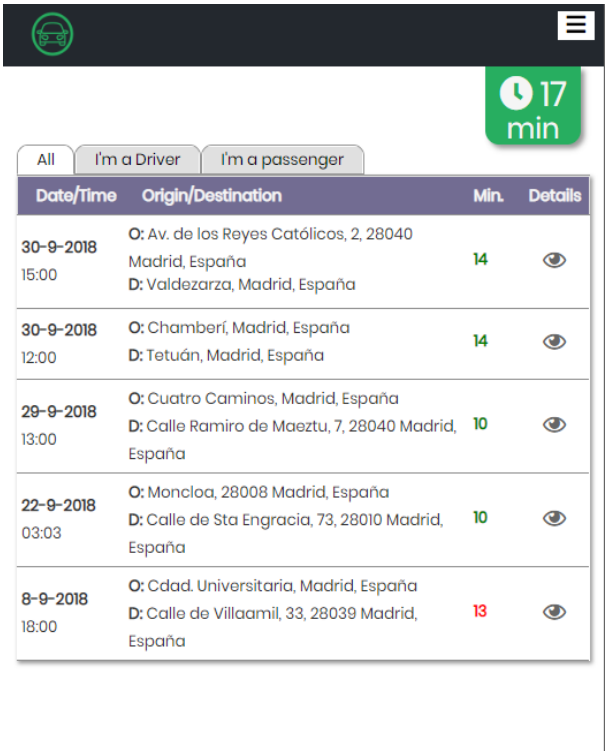

Figura A.11. Interfaz My Trips.

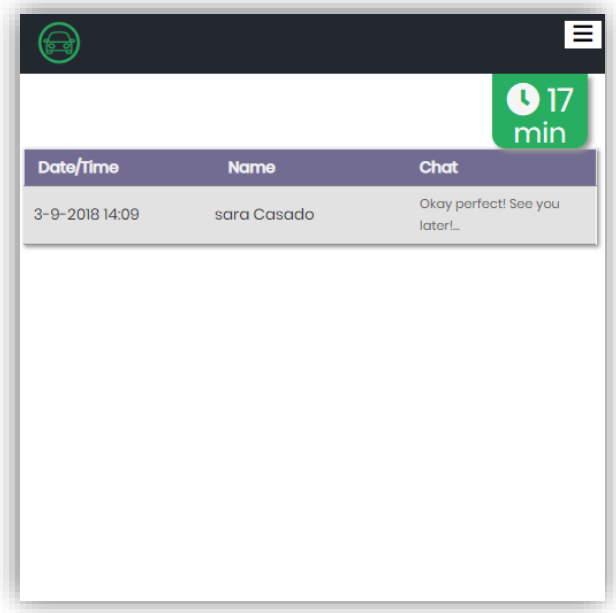

Figura A.12. Interfaz Inbox.

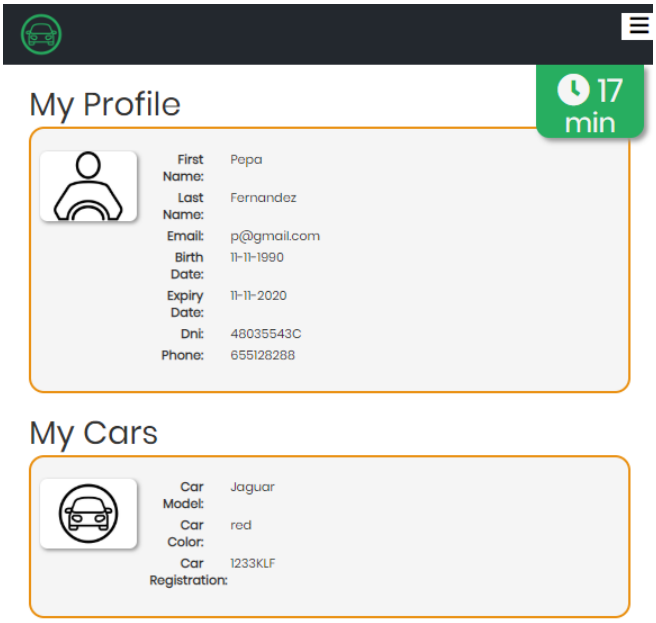

Figura A.13. Interfaz My Account,